

Desarrollo de Aplicaciones iOS

Índice

1 El entorno Xcode.....	4
1.1 Estructura del proyecto.....	8
1.2 Propiedades del proyecto.....	11
1.3 Configuraciones.....	15
1.4 Localización.....	16
1.5 Esquemas y acciones.....	18
1.6 Recursos y grupos.....	20
1.7 Interface Builder.....	22
1.8 Organizer.....	23
1.9 Repositorios SCM.....	25
1.10 Snapshots.....	32
1.11 Ejecución y firma.....	33
2 Ejercicios de Xcode.....	40
2.1 Creación de un proyecto con Xcode.....	40
2.2 Iconos y recursos.....	40
2.3 Localización.....	40
3 Introducción a Objective-C.....	42
3.1 Tipos de datos.....	42
3.2 Directivas.....	44
3.3 Paso de mensajes.....	48
3.4 Clases y objetos.....	49
3.5 Protocolos.....	60
3.6 Categorías y extensiones.....	61
3.7 Algunas clases básicas de Cocoa Touch.....	62
4 Ejercicios de Objective-C.....	74
4.1 Manejo de cadenas.....	74

4.2 Creación de objetos.....	74
4.3 Manejo de fechas (*).	75
4.4 Gestión de errores (*).	75
5 Propiedades, colecciones y gestión de la memoria.....	77
5.1 Propiedades de los objetos.....	77
5.2 Colecciones de datos.....	85
5.3 Key-Value-Coding (KVC).....	93
5.4 Programación de eventos.....	95
5.5 Introspección.....	100
5.6 Ciclo de vida de las aplicaciones.....	103
6 Ejercicios de propiedades y colecciones.....	105
6.1 Propiedades.....	105
6.2 Listas.....	105
6.3 Temporizadores (*).	105
6.4 Gestión de memoria con ARC.....	106
7 Vistas.....	107
7.1 Patrón Modelo-Vista-Controlador (MVC).....	107
7.2 Jerarquía de vistas, ventanas, y controladores.....	107
7.3 Creación de vistas con Interface Builder.....	110
7.4 Creación de vistas de forma programática.....	118
7.5 Propiedades de las vistas.....	122
7.6 Controles básicos.....	123
8 Ejercicios de vistas.....	126
8.1 Creación de la vista con Interface Builder.....	126
8.2 Conexión de componentes mediante outlets.....	127
8.3 Controles y acciones.....	128
8.4 Imágenes.....	129
8.5 Carga de componentes del NIB.....	129
8.6 Almacenamiento (*).	130
8.7 Creación programática de componentes (*).	131
9 Controladores.....	132
9.1 Creación de un controlador propio.....	132

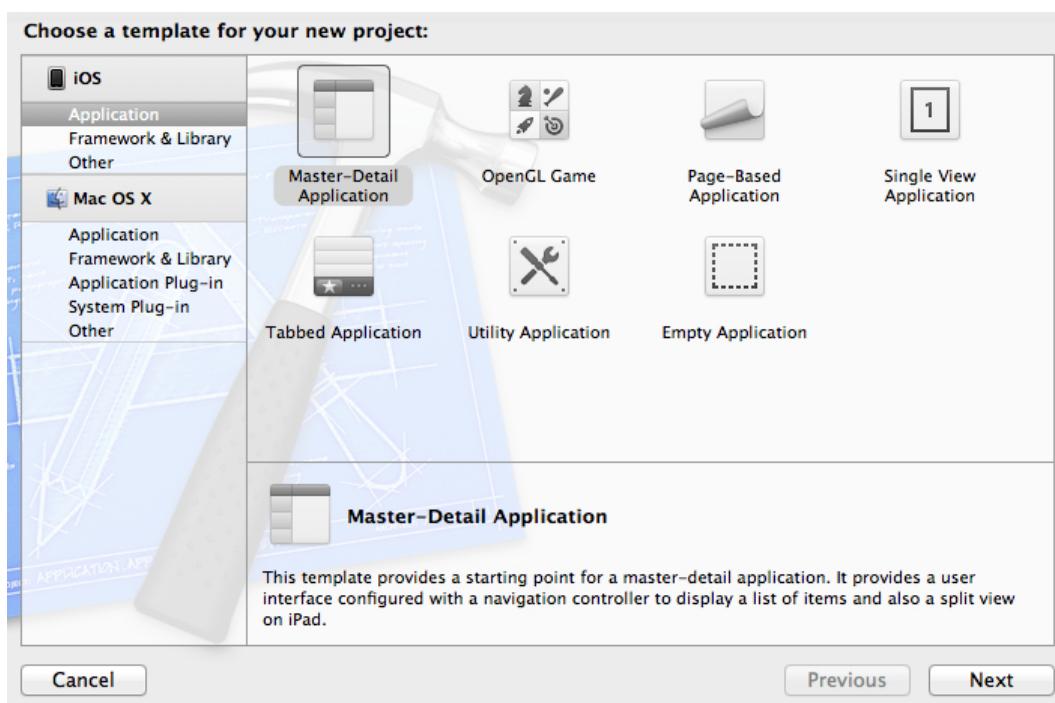
9.2 Controlador para las tablas.....	137
9.3 Controlador de búsqueda.....	147
10 Ejercicios de controladores.....	156
10.1 Creación de un controlador propio.....	156
10.2 Edición de tablas.....	157
10.3 Búsqueda (*).....	158
11 Transiciones y storyboards.....	160
11.1 Controladores modales.....	160
11.2 Controlador de navegación.....	161
11.3 Controlador de barra de pestañas.....	166
11.4 Uso de storyboards.....	168
12 Ejercicios de transiciones y storyboards.....	173
12.1 Pestañas.....	173
12.2 Aplicación basada en storyboards.....	174
13 Componentes para iPad y aplicaciones universales.....	181
13.1 Componentes específicos para iPad.....	181
13.2 Aplicaciones universales.....	204
14 Componentes para iPad y aplicaciones universales - Ejercicios.....	214
14.1 Programando una aplicación para iPad con Split View (1).....	214
14.2 Programando una aplicación para iPad con Split View (2).....	214
14.3 Programando una aplicación para iPad con Split View usando XCode 4.2.....	215
14.4 Programando una aplicación universal.....	215
14.5 Programando una aplicación para iPad con un Popover.....	216
15 Guías de estilo y personalizaciones avanzadas.....	217
15.1 Guías de estilo en iOS.....	217
15.2 Personalización avanzada: celdas.....	225
15.3 Personalización de ToolBars.....	231
16 Guías de estilo y personalizaciones avanzadas - Ejercicios.....	235
16.1 Personalizando las celdas de un Table View (1).....	235
16.2 Personalizando las celdas de un Table View (2).....	236
16.3 Personalizando un Tool Bar.....	236

1. El entorno Xcode

Para desarrollar aplicaciones iOS (iPod touch, iPhone, iPad) deberemos trabajar con una serie de tecnologías y herramientas determinadas. Por un lado, deberemos trabajar con el IDE Xcode dentro de una máquina con el sistema operativo MacOS instalado, cosa que únicamente podremos hacer en ordenadores Mac. Por otro lado, el lenguaje que deberemos utilizar es Objective-C, una extensión orientada a objetos del lenguaje C, pero muy diferente a C++. Además, deberemos utilizar la librería Cocoa Touch para crear una interfaz para las aplicaciones que pueda ser visualizada en un dispositivo iOS. Vamos a dedicar las primeras sesiones del módulo a estudiar fundamentalmente el entorno Xcode y el lenguaje Objective-C, pero introduciremos también algunos elementos básicos de Cocoa Touch para así poder practicar desde un primer momento con aplicaciones iOS.

Comenzaremos viendo cómo empezar a trabajar con el entorno Xcode. Vamos a centrarnos en la versión 4.2 de dicho IDE, ya que es la que se está utilizando actualmente y cuenta con grandes diferencias respecto a sus predecesoras.

Al abrir el entorno, lo primero que deberemos hacer es crear un nuevo proyecto con *File > New > New Project....*. En el asistente para la creación del proyecto nos dará a elegir una serie de plantillas de las que partir. En el lateral izquierdo vemos una serie de categorías de posibles plantillas, y en la parte central vemos las plantillas de la categoría seleccionada. Vemos además que las categorías se dividen en iOS y Mac OS X. Como es evidente, nos interesarán aquellas de la sección iOS.



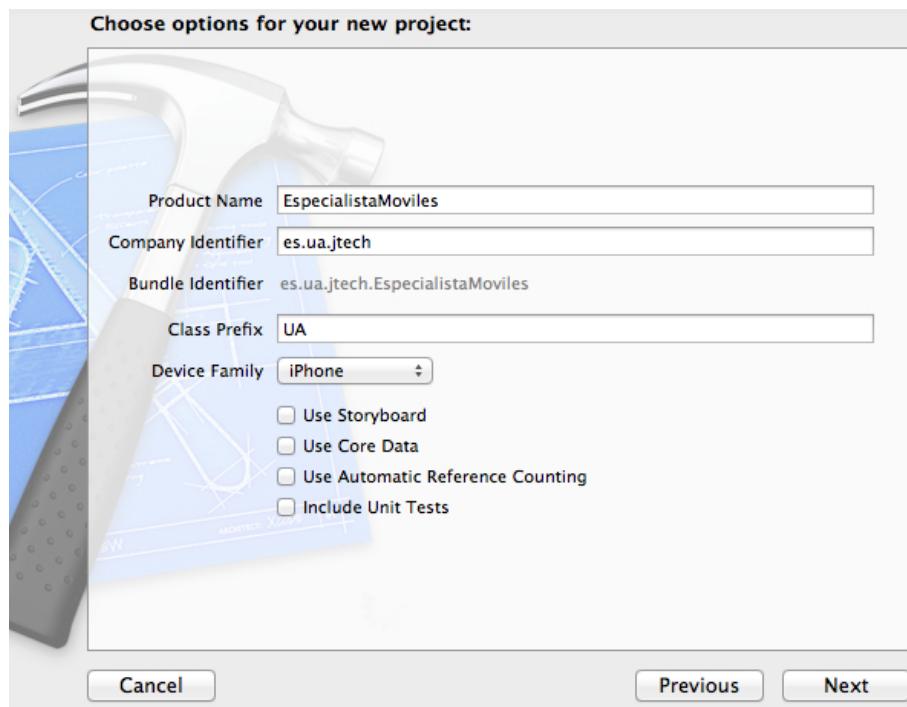
Tipos de proyectos

Las categorías que encontramos son:

- *Application*: Estos son los elementos que nos interesarán. Con ellos podemos crear diferentes plantillas de aplicaciones iOS. Encontramos plantillas para los estilos de navegación más comunes en aplicaciones iOS, que estudiaremos con mayor detenimiento más adelante, y también alguna plantilla básica como *Empty Application*, que no crea más componentes que una ventana vacía. Deberemos elegir aquella plantilla que más se ajuste al tipo de aplicación que vayamos a realizar. La más común en aplicaciones iOS es *Master-Detail Application* (en versiones anteriores de Xcode el tipo equivalente se llamaba *Navigation-based Application*), por lo que será la que utilicemos durante las primeras sesiones.
- *Framework & Library*: Nos permitirá crear librerías o *frameworks* que podamos utilizar en nuestras aplicaciones iOS. No se podrán ejecutar directamente como una aplicación, pero pueden ser introducidos en diferentes aplicaciones.
- *Other*: Aquí encontramos la opción de crear un nuevo proyecto vacío, sin utilizar ninguna plantilla.

Tras seleccionar un tipo de proyecto, nos pedirá el nombre del producto y el identificador de la compañía. El nombre del producto será el nombre que queramos darle al proyecto y a nuestra aplicación, aunque más adelante veremos que podemos modificar el nombre que se muestra al usuario. El identificador de la compañía se compone de una serie de cadenas en minúscula separadas por puntos que nos identificarán como desarrolladores. Normalmente pondremos aquí algo similar a nuestra URL escrita en orden inverso (como se hace con los nombres de paquetes en Java). Por ejemplo, si nuestra web es `jtech.ua.es`, pondremos como identificador de la compañía `es.ua.jtech`. Con el nombre del producto junto al identificador de la compañía compondrá el identificador del paquete (*bundle*) de nuestra aplicación. Por ejemplo, una aplicación nuestra con nombre `EspecialistaMoviles`, tendría el identificador `es.ua.jtech.EspecialistaMoviles`.

Podremos también especificar un prefijo para las clases de nuestra aplicación. En Objective-C no existen los paquetes como en lenguaje Java, por lo que como espacio de nombres para las clases es habitual utilizar un determinado prefijo. Por ejemplo, si estamos desarrollando una aplicación de la Universidad de Alicante, podríamos especificar como prefijo `UA`, de forma que todas las clases de nuestra aplicación llevarán ese prefijo en su nombre y nos permitirá distinguirlas de clases de librerías que estemos utilizando, y que podrían llamarse de la misma forma que las nuestras si no llevasen prefijo.



Datos del proyecto

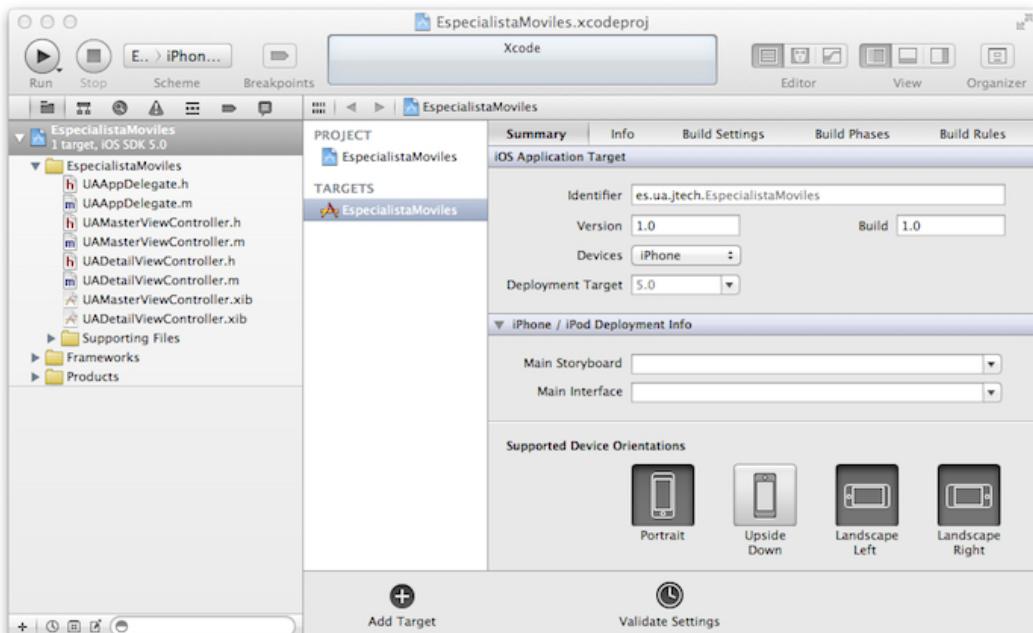
Bajo los campos anteriores, aparece un desplegable que nos permite elegir a qué tipo de dispositivos vamos a destinar la aplicación: iPhone, iPad, o Universal. Las aplicaciones de iPhone se pueden ver en el iPad emuladas, pero en un recuadro del tamaño del iPhone. Las de iPad sólo son compatibles con el iPad, mientras que las aplicaciones Universales son compatibles con ambos dispositivos y además su interfaz de adapta a cada uno de ellos. Por ahora comenzaremos realizando aplicaciones únicamente para iPhone, y más adelante veremos cómo desarrollar aplicaciones para iPad y aplicaciones Universales.

También tendremos cuatro *checkboxes* que para añadir *Storyboards*, *Core Data*, *Automatic Reference Counting* (ARC) y pruebas de unidad a nuestra aplicación. Por ahora vamos a desmarcarlos. Más adelante estudiaremos dichos elementos.

Tras llenar estos datos, pulsaremos *Next* y para finalizar tendremos que seleccionar la ubicación del sistema de archivos donde guardar el proyecto. Se creará un directorio con el nombre de nuestro proyecto en la localización que indiquemos, y dicho directorio contendrá todos los componentes del proyecto.

Nota

En la ventana de selección de la ubicación en la que guardar el proyecto nos dará también la opción de crear automáticamente un repositorio SCM local de tipo Git para dicho proyecto. Por el momento vamos a utilizar dicho repositorio, ya que lo único que tendremos que hacer es dejar marcada la casilla y Xcode se encargará de todo lo demás. Más adelante veremos como subir el proyecto a un repositorio remoto propio.



Aspecto del entorno Xcode

Una vez creado el proyecto, veremos la ventana principal del entorno Xcode. Podemos distinguir las siguientes secciones:

- **Navegador:** A la derecha vemos los artefactos de los que se compone el proyecto, organizados en una serie de grupos. Esta sección se compone de varias pestañas, que nos permitirán navegar por otros tipos de elementos, como por ejemplo por los resultados de una búsqueda, o por los resultados de la construcción del proyecto (errores y warnings).
- **Editor:** En la parte central de la pantalla vemos el editor en el que podremos trabajar con el fichero que esté actualmente seleccionado en el navegador. Dependiendo del tipo de fichero, editaremos código fuente, un fichero de propiedades, o la interfaz de la aplicación de forma visual.
- **Botones de construcción:** En la parte superior izquierda vemos una serie de botones que nos servirán para construir y ejecutar la aplicación.
- **Estado:** En la parte central de la parte superior veremos la información de estado del entorno, donde se mostrarán las acciones que se están realizando en un momento dado (como compilar o ejecutar el proyecto).
- **Opciones de la interfaz:** En la parte derecha de la barra superior tenemos una serie de botones que nos permitirán alterar el aspecto de la interfaz, mostrando u ocultando algunos elementos, como puede ser la vista de navegación (a la izquierda), la consola de depuración y búsqueda (abajo), y la barra de utilidades y ayuda rápida (derecha). También podemos abrir una segunda ventana de Xcode conocida como *Organizer*, que nos permitirá, entre otras cosas, gestionar los dispositivos con los que contamos o navegar por la documentación. Más adelante veremos dicha ventana con más detalle.

Recomendación

Resulta bastante útil mantener abierta la vista de ayuda rápida a la derecha de la pantalla, ya que conforme editamos el código aquí nos aparecerá la documentación de los elementos de la API sobre los que se encuentre el cursor.

En este momento podemos probar a ejecutar nuestro proyecto. Simplemente tendremos que pulsar el botón de ejecutar (en la parte superior izquierda), asegurándonos que junto a él en el cuadro desplegable tenemos seleccionado que se ejecute en el simulador del iPhone. Al pulsar el botón de ejecución, se mostrará el progreso de la construcción del proyecto en la parte superior central, y una vez finalizada se abrirá una ventana con un emulador de iPhone y se ejecutará nuestra aplicación en él, que actualmente será únicamente una pantalla con una barra de navegación y una lista vacía. En la barra de menús del simulador podemos encontrar opciones para simular diferentes condiciones del hardware, como por ejemplo los giros del dispositivo.



Simulador de iPhone

1.1. Estructura del proyecto

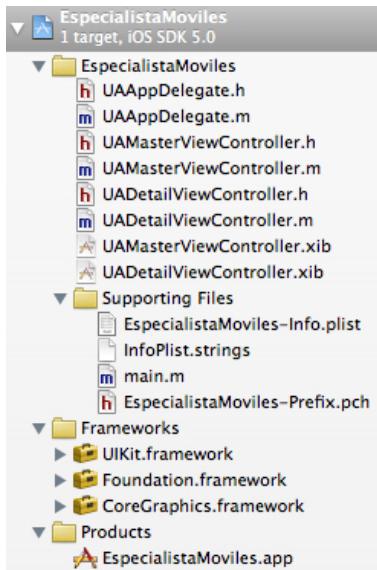
En la zona del navegador podemos ver los diferentes elementos que componen nuestro proyecto. Los tipos de ficheros que encontraremos normalmente en un proyecto Objective-C son los siguientes:

- .m: El código fuente Objective-C se guarda en ficheros con extensión .m, por lo que

ese será el formato que encontraremos normalmente, aunque también se podrá utilizar código C (.c) estándar, C++ (.cpp, .cc), u Objective-C combinado con C++ (.mm).

- .h: También tenemos los ficheros de cabeceras (.h). Las clases de Objective-C se componen de un fichero .m y un fichero .h con el mismo nombre.
- .xib: También encontramos .xib (también conocidos como nib, ya que este es su formato una vez compilados) que son los ficheros que contienen el código de la interfaz, y que en el entorno se editarán de forma visual.
- .plist: Formato de fichero de propiedades que estudiaremos más adelante con mayor detenimiento, y que se utiliza para almacenar una lista de propiedades. Resultan muy útiles para guardar los datos o la configuración de nuestra aplicación, ya que resultan muy fáciles de editar desde el entorno, y muy fáciles de leer y de escribir desde el programa.
- .strings: El formato .strings define un fichero en el que se definen una serie de cadenas de texto, cada una de ellas asociada a un identificador, de forma que creando varias versiones del mismo fichero .strings tendremos la aplicación localizada a diferentes idiomas.
- .app: Es el resultado de construir y empaquetar la aplicación. Este fichero es el que deberemos subir a iTunes para su publicación en la App Store.

Estos diferentes tipos de artefactos se pueden introducir en nuestro proyecto organizados por grupos. Es importante destacar que los grupos no corresponden a ninguna organización en el disco, ni tienen ninguna repercusión en la construcción del proyecto. Simplemente son una forma de tener los artefactos de nuestro proyecto organizados en el entorno, independientemente del lugar del disco donde residan realmente. Los grupos se muestran en el navegador con el icono de una carpeta amarilla.

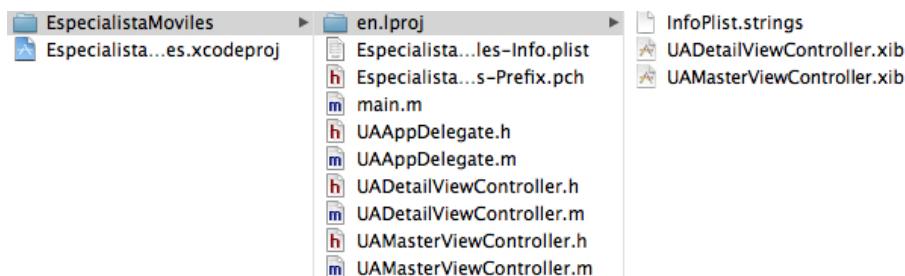


Configuración del proyecto creado

En la plantilla que nos ha creado encontramos los siguientes grupos de artefactos:

- **Fuentes** (están en un grupo con el mismo nombre del proyecto, en nuestro caso `EspecialistaMoviles`). Aquí se guardan todos los ficheros de código fuente que componen la aplicación (código fuente `.m`, cabeceras `.h` y ficheros de la interfaz `.xib`).
- **Soporte** (dentro del grupo de fuentes, en un subgrupo *Supporting files*). Aquí encontramos algunos recursos adicionales, como el fichero `main.m`, que es el punto de arranque de la aplicación, y que nosotros normalmente no deberemos tocar, el fichero `Proyecto-Prefix.pch`, que define un prefijo con una serie de *imports* que se aplicarán a todos los ficheros del proyecto (normalmente para importar la API de Cocoa Touch), `Proyecto-Info.plist`, que define una serie de propiedades de la aplicación (nombre, identificador, versión, ícono, tipos de dispositivos soportados, etc), y por último `InfoPlist.strings`, donde se externalizan las cadenas de texto que se necesitan utilizar en el fichero `.plist` anterior, para así poder localizar la aplicación fácilmente.
- **Frameworks**: Aquí se muestran los *frameworks* que la aplicación está utilizando actualmente. Por defecto nos habrá incluido `UIKit`, con la API de la interfaz de Cocoa Touch, `Foundation`, con la API básica de propósito general (cadenas, *arrays*, fechas, URLs, etc), y `CoreGraphics`, con la API básica de gráficos (fuentes, imágenes, geometría).
- **Products**: Aquí vemos los resultados de construir la aplicación. Veremos un fichero `EspecialistaMoviles.app` en rojo, ya que dicho fichero todavía no existe, es el producto que se generará cuando se construya la aplicación. Este fichero es el paquete (*bundle*) que contendrá la aplicación compilada y todos los recursos y ficheros de configuración necesarios para su ejecución. Esto es lo que deberemos enviar a Apple para su publicación en la App Store una vez finalizada la aplicación.

Sin embargo, si nos fijamos en la estructura de ficheros que hay almacenada realmente en el disco podemos ver que dichos grupos no existen:



Estructura de ficheros del proyecto

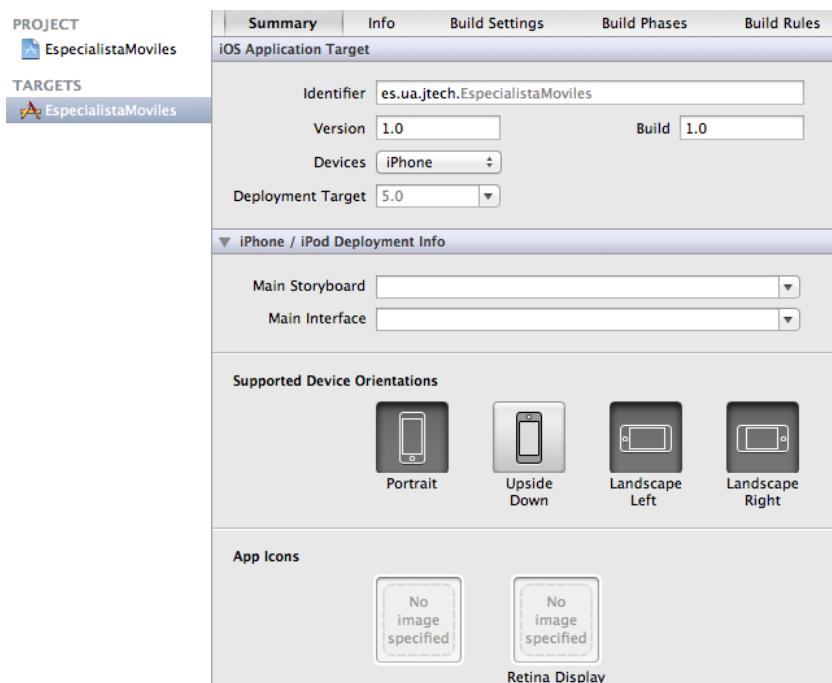
Vemos que tenemos en el directorio principal un fichero con extensión `xcodeproj`. Este es el fichero con la configuración de nuestro proyecto, y es el que deberemos seleccionar para abrir nuestro proyecto la próxima vez que vayamos a trabajar con él (haciendo doble-click sobre él se abrirá Xcode con nuestro proyecto). A parte de este fichero, tenemos un directorio con el nombre del proyecto que contiene todos los ficheros al mismo nivel. La única excepción son los ficheros que hay en un directorio de nombre

`en.lproj`. Esto se debe a que dichos ficheros están localizados, concretamente a idioma inglés como indica su directorio (`en`). Tendremos un subdirectorio como este por cada idioma que soporte nuestra aplicación (`es.lproj`, `ca.lproj`, etc). Es la forma en la que se trata la localización en las aplicaciones iOS, y lo veremos con más detalle a continuación.

1.2. Propiedades del proyecto

Si seleccionamos en el navegador el nodo raíz de nuestro proyecto, en el editor veremos una ficha con sus propiedades. Podemos distinguir dos apartados: *Project* y *Targets*. En *Project* tenemos la configuración general de nuestro proyecto, mientras que *Target* hace referencia a la construcción de un producto concreto. En nuestro caso sólo tenemos un *target*, que es la construcción de nuestra aplicación, pero podría haber varios. Por ejemplo, podríamos tener un target que construya nuestra aplicación para iPhone y otro que lo haga para iPad, empaquetando en cada una de ellas distintos recursos y compilando con diferentes propiedades. También tendremos dos *targets* si hemos creado pruebas de unidad, ya que tendremos un producto en el que se incluirán las pruebas, y otro en el que sólo se incluirá el código de producción que será el que publicaremos.

Si pinchamos sobre el *target*, en la pantalla principal veremos una serie de propiedades de nuestra aplicación que resultan bastante intuitivas, como la versión, dispositivos a los que está destinada, y versión mínima de iOS que necesita para funcionar. También podemos añadir aquí tanto los iconos como las imágenes para la pantalla de carga de nuestra aplicación, simplemente arrastrando dichas imágenes sobre los huecos correspondientes.



Configuración básica del target

Lo que no resulta tan claro es por ejemplo la diferencia entre versión y *build*. La primera de ellas indica el número de una *release* pública, y es lo que verá el usuario en la App Store, con el formato de tres enteros separados por puntos (por ejemplo, 1.2.1). Sin embargo, la segunda se utiliza para cada *build* interna. No es necesario seguir ningún formato dado, y puede coincidir o no con el número de versión. En muchas ocasiones se utiliza un número entero de *build* que vamos incrementando continuamente con cada iteración del proyecto, por ejemplo 1524. Otra diferencia existente entre ambos números es que podemos tener un número diferente de versión para distintos *targets*, pero la *build* siempre será la misma para todos ellos.

También vemos unos campos llamados *Main Storyboard* y *Main Interface*. Aquí se puede definir el punto de entrada de nuestra aplicación de forma declarativa, pudiendo especificar qué interfaz (fichero *xib*) o que *storyboard* se va a mostrar al arrancar la aplicación. En la plantilla crear ninguno de estos campos tiene valor, ya que la interfaz se crea de forma programática. En realidad el punto de entrada está en *main.m*, y podríamos modificarlo para tener mayor control sobre lo que la aplicación carga al arrancar, pero normalmente lo dejaremos tal como se crea por defecto. Más adelante veremos más detalles sobre el proceso de carga de la aplicación.

En la pestaña *Info* del *target* vemos los elementos de configuración anteriores, y algunos más, pero en forma de lista de propiedades. Realmente esta información es la que se almacena en el fichero *Info.plist*, si seleccionamos dicho fichero veremos los mismos datos también presentados como lista de propiedades. Por ejemplo la propiedad *Bundle display name* no estaba en la pantalla anterior, y nos permite especificar el nombre que aparecerá bajo el ícono de nuestra aplicación en el iPhone. Por defecto tenemos el nombre del proyecto, pero podemos modificarlo por ejemplo para acortarlo y así evitar que el iPhone recorte las letras que no quepan. Si pinchamos con el botón derecho sobre el fichero *Info.plist* y seleccionamos *Open As > Source Code*, veremos el contenido de este fichero en el formato XML en el que se definen los ficheros de listas de propiedades, y los nombres internos que tiene cada una de las propiedades de configuración.

PROJECT	Summary	Info	Build Settings	Build Phases	Build Rules
EspecialistaMovies					
TARGETS					
EspecialistaMovies					
Custom iOS Target Properties					
	Key	Type	Value		
	Bundle name	String	\$(PRODUCT_NAME)		
	Bundle identifier	String	es.ua.tech.\$[PRODUCT_NAME]identifier		
	InfoDictionary version	String	6.0		
	► Required device capabilities	Array	(1 item)		
	Bundle version	String	1.0		
	Executable file	String	\$(EXECUTABLE_NAME)		
	Application requires iPhone environment	Boolean	YES		
	► Icon files	Array	(0 items)		
	► Supported interface orientations	Array	(3 items)		
	Bundle display name	String	\$(PRODUCT_NAME)		
	Bundle OS Type code	String	APPL		
	Bundle creator OS Type code	String	????		
	Localization native development region	String	en		
	Bundle versions string, short	String	1.0		
	► Document Types (0)				
	► Exported UTIs (0)				
	► Imported UTIs (0)				
	► URL Types (0)				

Configuración completa del target

A continuación tenemos la pestaña *Build Settings*. Aquí se define una completa lista con todos los parámetros utilizados para la construcción del producto. Estos parámetros se pueden definir a diferentes niveles: valores por defecto para todos los proyectos iOS, valores generales para nuestro proyecto, y valores concretos para el *target* seleccionado. Se pueden mostrar los valores especificados en los diferentes niveles, y el valor que se utilizará (se coge el valor más a la izquierda que se haya introducido):

PROJECT	Summary	Info	Build Settings	Build Phases	Build Rules
EspecialistaMovies					
TARGETS					
EspecialistaMovies					
Basic All Combined Levels					
Setting	Resolved	Especialista... Especialista... iOS Default			
Architectures					
Additional SDKs					
Architectures	Standard	Standard	armv7		
Base SDK	Latest iOS (i...)	Latest iOS (i...)	iOS 5.0		
Build Active Architecture Only	No	No			
Supported Platforms	iphonesimulat...	iphonesimulat...			
Valid Architectures	armv6 armv7	armv6 armv7			
Build Locations					
Build Products Path	/Users/maloza...		/Users/maloza...		
Intermediate Build Files Path	/Users/maloza...		/Users/maloza...		
► Per-configuration Build Products Path	<Multiple val...		<Multiple val...		
► Per-configuration Intermediate Build Fi...	<Multiple val...		<Multiple val...		
Precompiled Headers Cache Path	/var/folders/p...		/var/folders/p...		
Build Options					
Build Variants	normal	normal			
Compiler for C/C++/Objective-C	Apple LLVM...	Apple LLVM...			
Debug Information Format	DWARF with...	DWARF with...			
Enable OpenMP Support	No	No			
Generate Profiling Code	No	No			
Precompiled Header Uses Files From B...	Yes	Yes			
Run Static Analyzer	No	No			
Scan All Source Files for Includes	No	No			
► Validate Built Product	<Multiple val... +	<Multiple val... +	No		

Parámetros de construcción del target

Hay un gran número de parámetros, por lo que es complicado saber lo que hace

exactamente cada uno, y la mayor parte de ellos no será necesario que los modifiquemos en ningún momento. Normalmente ya se les habrá asignado por defecto un valor adecuado, por lo que aquí la regla de oro es **si no sabes lo que hace, ¡no lo toques!**. En esta interfaz podremos cambiar los valores para el *target*. Si queremos cambiar los valores a nivel del proyecto, deberemos hacerlo seleccionando el proyecto en el lateral izquierdo del editor y accediendo a esta misma pestaña.

La información más importante que podemos encontrar aquí es:

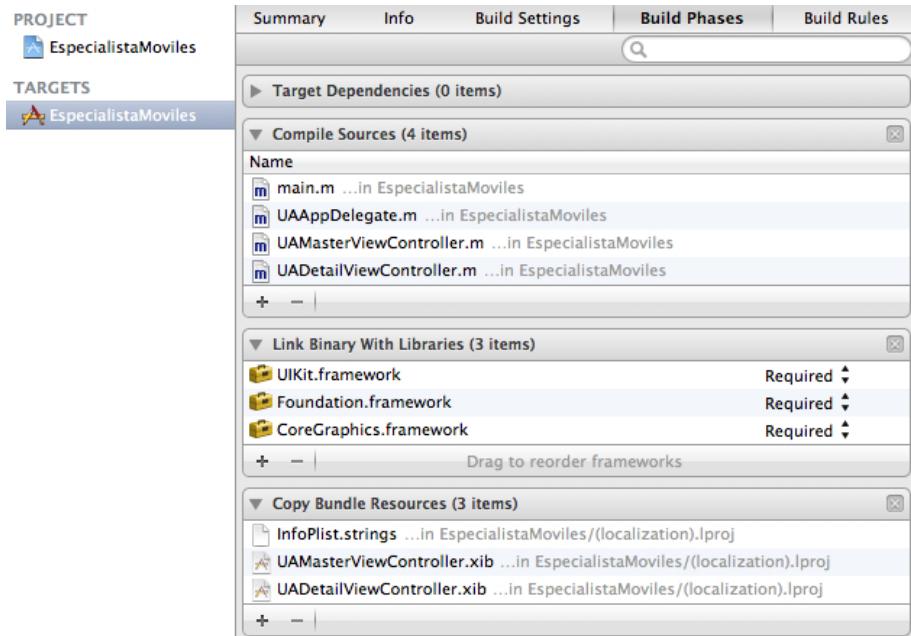
- *Base SDK*: Versión del SDK que se utiliza para compilar. Se diferencia de *Deployment Target* en que *Base SDK* indica la versión del SDK con la que se compila el proyecto, por lo que no podremos utilizar ninguna característica que se hubiese incluido en versiones posteriores a la seleccionada, mientras que *Deployment Target* indica la versión mínima para que nuestra aplicación funcione, independientemente de la versión con la que se haya compilado.
- *Code Signing*: Aquí podemos especificar el certificado con el que se firma la aplicación para probarla en dispositivos reales o para distribuirla. Más adelante veremos con más detalle la forma de obtener dicho certificado. Hasta que no lo tengamos, no podremos probar las aplicaciones más que en el emulador.

Advertencia

Si seleccionamos un *Deployment Target* menor que el *Base SDK*, deberemos llevar mucho cuidado de no utilizar ninguna característica que se haya incluido en versiones posteriores al *Deployment Target*. Si no llevásemos cuidado con esto, un comprador con una versión antigua de iOS podría adquirir nuestra aplicación pero no le funcionaría. Lo que si que podemos hacer es detectar en tiempo de ejecución si una determinada característica está disponible, y sólo utilizarla en ese caso.

Vemos también que para algunas opciones tenemos dos valores, uno para *Debug* y otro para *Release*. Estas son las configuraciones del proyecto, que veremos más adelante. Por ejemplo, podemos observar que para *Debug* se añade una macro `DEBUG` y se elimina la optimización del compilador para agilizar el proceso de construcción y hacer el código más fácilmente depurable, mientras que para *Release* se eliminan los símbolos de depuración para reducir el tamaño del binario.

La siguiente pestaña (*Build Phases*) es la que define realmente cómo se construye el *target* seleccionado. En ella figuran las diferentes fases de construcción del proyecto, y para cada una de ellas indica los ficheros que se utilizarán en ella:



Fases de construcción del target

Podemos ver aquí los ficheros que se van a compilar, las librerías con las que se va a enlazar, y los recursos que se van a copiar dentro del *bundle*. Por defecto, cuando añadamos elementos al proyecto se incluirán en la fase adecuada (cuando sea código se añadirá a la compilación, los *frameworks* a *linkado*, y el resto de recursos a copia). Sin embargo, en algunas ocasiones puede que algún fichero no sea añadido al lugar correcto, o que tengamos varios *targets* y nos interese controlar qué recursos se incluyen en cada uno de ellos. En esos casos tendremos que editar a mano esta configuración. Es importante destacar que el contenido y estructura final del paquete (*bundle*) generado se define en esta pantalla, y es totalmente independiente a la organización del proyecto que vemos en el navegador.

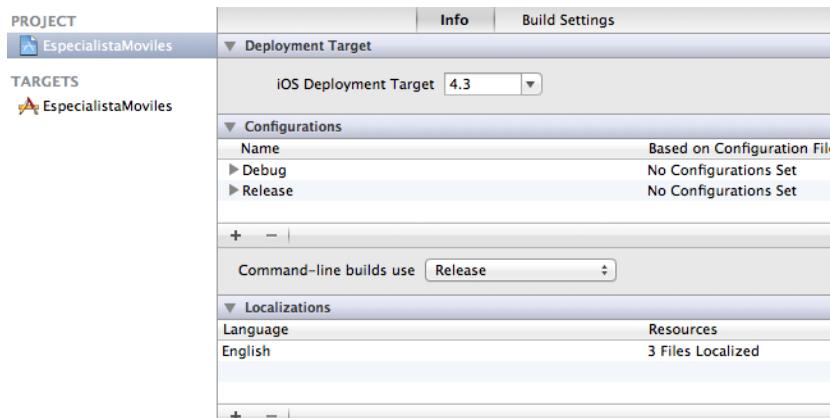
Por último, en *Build Rules*, se definen las reglas para la construcción de cada tipo de recurso. Por ejemplo, aquí se indica que las imágenes o ficheros de propiedades simplemente se copiarán, mientras que los ficheros .xib deberán compilarse con el compilador de Interface Builder. Normalmente no tendremos que modificar esta configuración.

1.3. Configuraciones

Hemos visto que en la pantalla *Build Settings* en algunos de los parámetros para la construcción de la aplicación teníamos dos valores: *Debug* y *Release*. Éstas son las denominadas configuraciones para la construcción de la aplicación. Por defecto se crean estas dos, pero podemos añadir configuraciones adicionales si lo consideramos oportuno (normalmente siempre se añade una tercera configuración para la distribución de la

aplicación, que veremos más adelante).

Podemos gestionar las configuraciones en la pestaña *Info* de las propiedades generales del proyecto:



Configuración general del proyecto

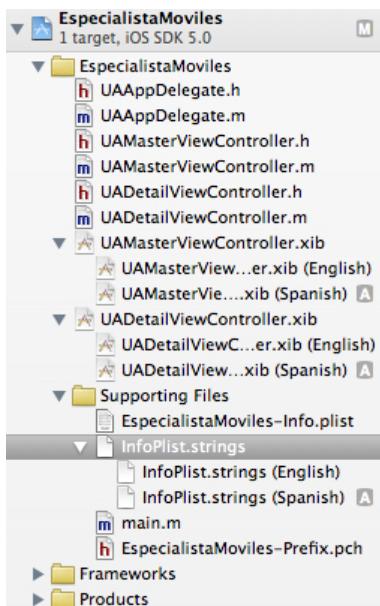
Podemos ver en la sección *Configurations* las dos configuraciones creadas por defecto. Bajo la lista, encontramos los botones (+) y (-) que nos permitirán añadir y eliminar configuraciones. Cuando añadimos una nueva configuración lo hacemos duplicando una de las existentes. Una práctica habitual es duplicar la configuración *Release* y crear una nueva configuración *Distribution* en la que cambiamos el certificado con el que se firma la aplicación.

Es importante diferenciar entre *targets* y *configuraciones*. Los *targets* nos permiten construir productos distintos, con diferentes nombre, propiedades, y conjuntos de fuentes y de recursos. Las configuraciones son diferentes formas de construir un mismo producto, cambiando las opciones del compilador, o utilizando diferentes firmas.

1.4. Localización

En la pantalla de propiedades generales del proyecto también vemos la posibilidad de gestionar las localizaciones de nuestra aplicación, es decir, los distintos idiomas en los que estará disponible. Por defecto sólo está en inglés, aunque con los botones (+) y (-) que hay bajo la lista podemos añadir fácilmente todas aquellas que queramos. Por ejemplo, podemos localizar también nuestra aplicación al español.

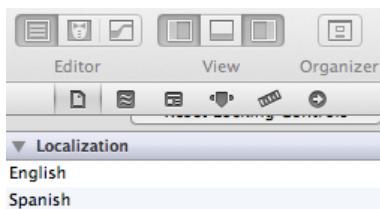
Por defecto localiza automáticamente los ficheros *xib* de la interfaz y los ficheros *strings* en los que se externalizan las cadenas de la aplicación. En el navegador veremos dos versiones de estos ficheros:



Ficheros localizados

En el disco veremos que realmente lo que se ha hecho es guardar dos copias de estos ficheros en los directorios `en.lproj` y `es.lproj`, indicando así la localización a la que pertenece cada uno.

Es posible, por ejemplo, que una interfaz no necesite ser localizada, bien por que no tiene textos, o porque los textos se van a poner desde el código. En tal caso, no es buena idea tener el fichero de interfaz duplicado, ya que cada cambio que hagamos en ella implicará modificar las dos copias. La gestión de la localización para cada fichero independiente se puede hacer desde el panel de utilidades (el panel de la derecha, que deberemos abrir en caso de que no esté abierto), seleccionando en el navegador el fichero para el que queremos modificar la localización:



Panel de utilidades de localización

De la misma forma, podemos también añadir localizaciones para ficheros que no estuviesen localizados. Por ejemplo, si tenemos una imagen en la que aparece algún texto, desde el panel de utilidades anterior podremos añadir una nueva localización y así tener una versión de la imagen en cada idioma.

Vemos también que por defecto nos ha creado (y localizado) el fichero

`InfoPlist.strings` que nos permite localizar las propiedades de `Info.plist`. Por ejemplo imaginemos que queremos que el nombre bajo el icono de la aplicación cambie según el idioma. Esta propiedad aparece en `Info.plist` como *Bundle display name*, pero realmente lo que necesitamos es el nombre interno de la propiedad, no la descripción que muestra Xcode de ella. Para ver esto, una vez abierto el fichero `Info.plist` en el editor, podemos pulsar sobre él con el botón derecho y seleccionar *Show Raw Keys/Values*. De esa forma veremos que la propiedad que buscamos se llama `CFBundleDisplayName`. Podemos localizarla en cada una de las versiones del fichero `InfoPlist.strings` de la siguiente forma:

```
// Versión en castellano del fichero InfoPlist.strings
"CFBundleDisplayName" = "MovilesUA";

// Versión en inglés del fichero InfoPlist.strings
"CFBundleDisplayName" = "MobileUA";
```

Si ejecutamos esta aplicación en el simulador, y pulsamos el botón *home* para volver a la pantalla principal, veremos que como nombre de nuestra aplicación aparece *MobileUA*. Si ahora desde la pantalla principal entramos en *Settings > General > International > Language* y cambiamos el idioma del simulador a español, veremos que el nombre de nuestra aplicación cambia a *MovilesUA*.

Con esto hemos localizado las cadenas de las propiedades del proyecto, pero ese fichero sólo se aplica al contenido de `Info.plist`. Si queremos localizar las cadenas de nuestra aplicación, por defecto se utilizará un fichero de nombre `Localizable.strings`, que seguirá el mismo formato (cada cadena se pone en una línea del tipo de las mostradas en el ejemplo anterior: `"identificador" = "cadena a mostrar" ;`).

Podemos crear ese fichero `strings` pulsando sobre el botón derecho del ratón sobre el grupo del navegador en el que lo queramos incluir y seleccionando *New File ...*. Como tipo de fichero seleccionaremos *iOS > Resource > Strings File*, y le llamaremos `Localizable`. No deberemos olvidar añadir las localizaciones desde el panel de utilizadas. Más adelante veremos cómo acceder desde el código de la aplicación a estas cadenas localizadas.

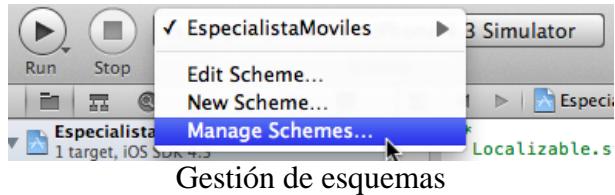
1.5. Esquemas y acciones

Otro elemento que encontramos para definir la forma en la que se construye y ejecuta la aplicación son los esquemas. Los esquemas son los encargados de vincular los *targets* a las configuraciones para cada una de las diferentes acciones que podemos realizar con el producto (construcción, ejecución, pruebas, análisis estático, análisis dinámico, o distribución).

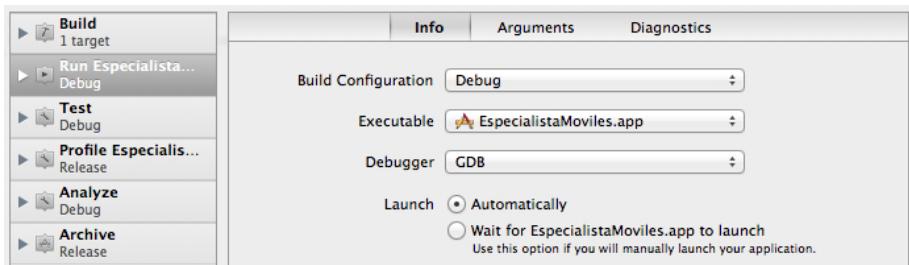
Por defecto nos habrá creado un único esquema con el nombre de nuestro proyecto, y normalmente no será necesario modificarlo ni crear esquemas alternativos, aunque vamos a ver qué información contiene para comprender mejor el proceso de construcción de la

aplicación.

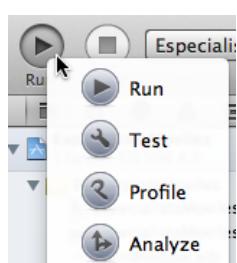
Para gestionar los esquemas debemos pulsar en el cuadro desplegable junto a los botones de ejecución de la aplicación:



Si queremos editar el esquema actual podemos pulsar en *Edit Scheme...*:



Vemos que el esquema contiene la configuración para cada posible acción que realicemos con el producto. La acción *Build* es un caso especial, ya que para el resto de acciones siempre será necesario que antes se construya el producto. En esta acción se especifica el *target* que se va a construir. En el resto de acciones se especificará la configuración que se utilizará para la construcción del target en cada caso. Podemos ejecutar estas diferentes acciones manteniendo pulsado el botón izquierdo del ratón sobre el botón *Run*:



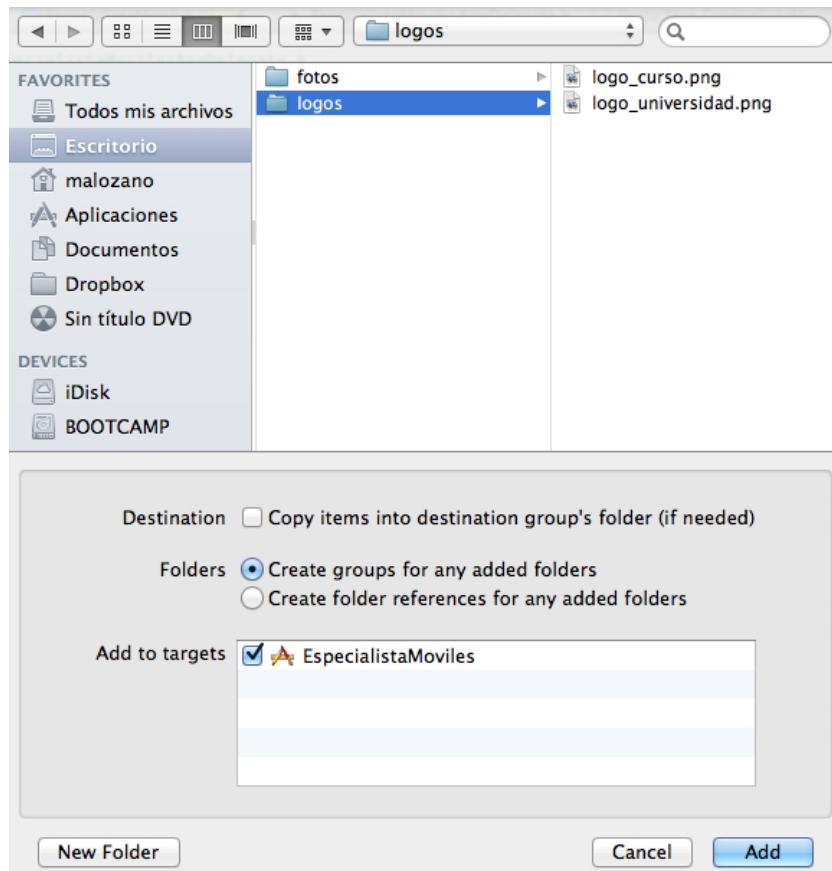
También pueden ser ejecutadas desde el menú *Product*, donde encontramos también la acción *Archive*, y la posibilidad de simplemente construir el producto sin realizar ninguna de las acciones anteriores. Las acciones disponibles son:

- *Build*: Construye el producto del *target* seleccionado en el esquema actual.
- *Run*: Ejecuta la aplicación en el dispositivo seleccionado (simulador o dispositivo real). Por defecto se construye con la configuración *Debug*.

- *Test*: Ejecuta las pruebas de unidad definidas en el producto. No funcionará si no hemos creado pruebas de unidad para el *target* seleccionado. Por defecto se construye con la configuración *Debug*.
- *Profile*: Ejecuta la aplicación para su análisis dinámico con la herramienta *Instruments*, que nos permitirá controlar en tiempo de ejecución elementos tales como el uso de la memoria, para poder así optimizarla y detectar posibles fugas de memoria. Más adelante estudiaremos esta herramienta con más detalles. Por defecto se construye con la configuración *Release*, ya que nos interesa optimizarla en tiempo de ejecución.
- *Analyze*: Realiza un análisis estático del código. Nos permite detectar construcciones de código incorrectas, como sentencias no alcanzables o posibles fugas de memoria desde el punto de vista estático. Por defecto se utiliza la configuración *Debug*.
- *Archive*: Genera un archivo para la distribución de nuestra aplicación. Para poder ejecutarla deberemos seleccionar como dispositivo destino *iOS Device*, en lugar de los simuladores. Utiliza por defecto la configuración *Release*, ya que es la versión que será publicada.

1.6. Recursos y grupos

Si queremos añadir recursos a nuestro proyecto, podemos pulsar con el botón derecho del ratón (*Ctrl-Click*) sobre el grupo en el que los queramos añadir, y seleccionar la opción *Add Files To "NombreProyecto"* Tendremos que seleccionar los recursos del disco que queramos añadir al proyecto. Puede ser un único fichero, o un conjunto de ficheros, carpetas y subcarpetas.



Añadir un nuevo recurso

Bajo el cuadro de selección de fichero vemos diferentes opciones. La primera de ellas es un *checkbox* con la opción *Copy items into destination group's folder (if needed)*. Si lo seleccionamos, copiará los recursos que añadamos el directorio en el que está el proyecto (en caso de que no estén ya dentro). En caso contrario, simplemente añadirá una referencia a ellos. Resulta conveniente copiarlos para que así el proyecto sea autocontenido, y podamos transportarlo fácilmente sin perder referencias.

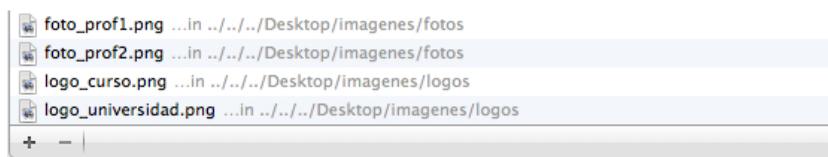
Bajo esta opción, nos da dos alternativas para incluir los ficheros seleccionados en el proyecto:

- *Create groups for any added folders*: Transforma las carpetas que se hayan seleccionado en grupos. Tendremos la misma estructura de grupos que la estructura de carpetas seleccionadas, pero los grupos sólo nos servirán para tener los recursos organizados dentro del entorno. Los grupos aparecerán como iconos de carpetas amarillas en el navegador.



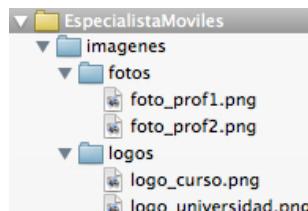
Grupo de recursos

Realmente todos los ficheros se incluirán en la raíz del paquete generado. Esto podemos comprobarlo si vamos a la sección *Build Phases* del *target* en el que hayamos incluido el recurso y consultamos la sección *Copy Bundle Resources*.



Copia de recursos en grupos

- *Create folder references for any added folders:* Con esta segunda opción las carpetas seleccionadas se incluyen como carpetas físicas en nuestro proyecto, no como grupos. Las carpetas físicas aparecen con el icono de carpeta de color azul.



Carpeta de recursos

En este caso la propia carpeta (con todo su contenido) será incluida en el paquete (*bundle*) generado. Si consultamos *Build Phases* veremos que lo que se copia al paquete en este caso es la carpeta completa.



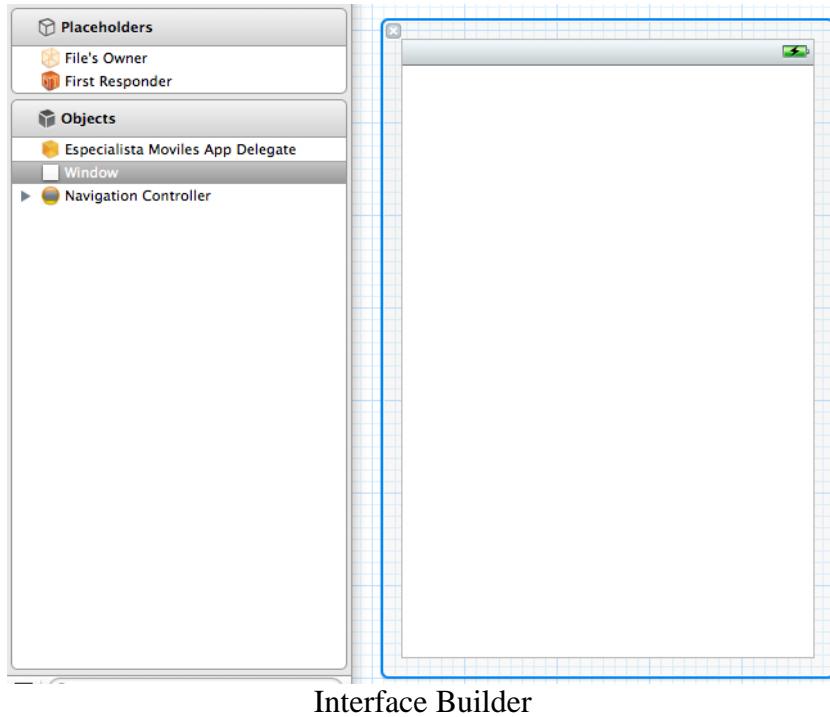
Copia de recursos en carpetas

Todos los recursos añadidos pueden ser localizados también como hemos visto anteriormente.

1.7. Interface Builder

Si en el navegador seleccionamos un fichero de tipo *xib*, en los que se define la interfaz, en el editor se abrirá la herramienta Interface Builder, que nos permitirá editar dicha

interfaz de forma visual. En versiones anteriores de Xcode Interface Builder era una herramienta independiente, pero a partir de Xcode 4 se encuentra integrado en el mismo entorno.

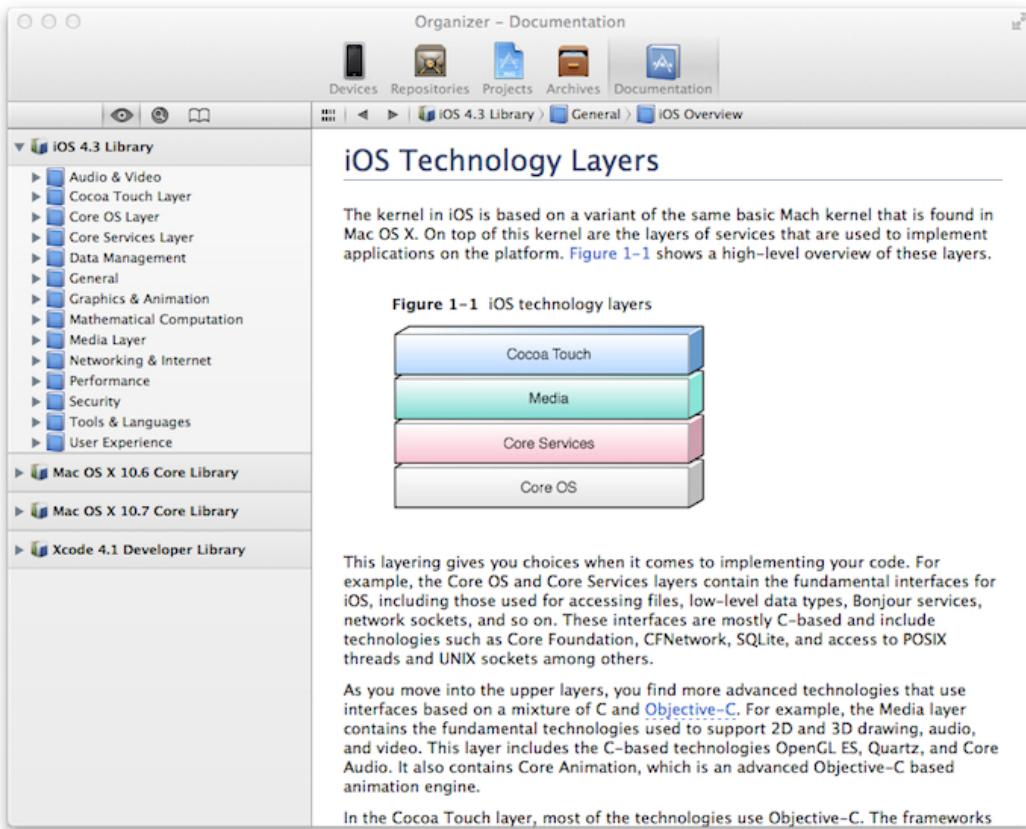


Veremos esta herramienta con más detalle cuando estudiemos la creación de la interfaz de las aplicaciones iOS.

1.8. Organizer

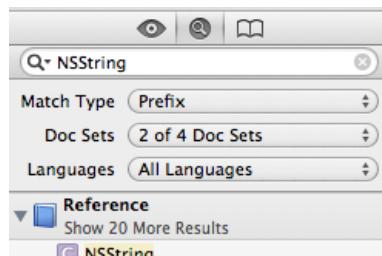
El *organizer* es una ventana independiente de la interfaz principal de Xcode desde donde podemos gestionar diferentes elementos como los dispositivos reales de los que disponemos para probar la aplicación, los repositorios de control de fuentes (SCM) que utilizamos en los proyectos, los archivos generados para la publicación de aplicaciones, y la documentación sobre las herramientas y las APIs con las que contamos en la plataforma iOS.

Podemos abrir *organizer* pulsando sobre el botón en la esquina superior derecha de la ventana principal de Xcode.



Ventana del Organizer

El uso más frecuente que le daremos el *organizer* será el de consultar la documentación de la plataforma iOS. La navegación por la documentación se hará de forma parecida a como se haría en un navegador. Normalmente nos interesarán buscar un determinado ítem. Para ello, en la barra lateral izquierda podemos especificar los criterios de búsqueda:



Búsqueda en la documentación

Es recomendable abrir los criterios avanzados de búsqueda y en *Doc Sets* eliminar los conjuntos de documentación de Mac OS X, ya que no nos interesarán para el desarrollo en iOS y así agilizaremos la búsqueda.

1.9. Repositorios SCM

Anteriormente hemos visto que Xcode nos pone muy fácil crear un repositorio Git para nuestro proyecto, ya que basta con marcar una casilla durante la creación del proyecto. Sin embargo, si queremos utilizar un repositorio remoto deberemos realizar una configuración adicional. Las opciones para trabajar con repositorios en Xcode son bastante limitadas, y en muchas ocasiones resulta poco intuitiva la forma de trabajar con ellas. Vamos a ver a continuación cómo trabajar con repositorios Git y SVN remotos.

Para gestionar los repositorios deberemos ir a la pestaña *Repositories* del *Organizer*. Ahí veremos en el lateral izquierdo un listado de los repositorios con los que estamos trabajando. Si hemos marcado la casilla de utilizar Git al crear nuestros proyectos, podremos ver aquí la información de dichos repositorios y el historial de revisiones de cada uno de ellos.

1.9.1. Repositorios Git

Si hemos creado un repositorio Git local para nuestro proyecto, será sencillo replicarlo en un repositorio Git remoto. En primer lugar, necesitaremos crear un repositorio remoto. Vamos a ver cómo crear un repositorio privado en BitBucket (bitbucket.org).

- En primer lugar, deberemos crearnos una cuenta en BitBucket, si no disponemos ya de una.
- Creamos desde nuestra cuenta de BitBucket un repositorio (*Repositories > Create repository*).
- Deberemos darle un nombre al repositorio. Será de tipo Git y como lenguaje especificaremos Objective-C.

The screenshot shows the BitBucket interface for creating a new repository. At the top, there's a navigation bar with the BitBucket logo, the word 'Atlassian', and tabs for 'Explore', 'Dashboard', and 'Repositories'. Below this is a dark blue header bar with the text 'Create new repository' and 'Start from scratch.' In the main form area, there are several input fields and options:

- Owner:** A dropdown menu showing 'malozano'.
- Name (required):** An input field containing 'Prueba'. To its right are three small icons: a green square with a white star, a red square with a white plus sign, and a yellow square with a white minus sign. Next to the name is a checked checkbox labeled 'Private'.
- Repository type:** A radio button group where 'Git' is selected, and 'Mercurial' is unselected.
- Project management:** A group of checkboxes where 'Issue tracking' is unselected and 'Wiki' is unselected.
- Language:** A dropdown menu showing 'Objective-C'.
- Description:** A large text area for entering a description, which is currently empty.
- Website:** A text input field for a website URL, which is also empty.
- Create repository:** A button at the bottom left of the form.

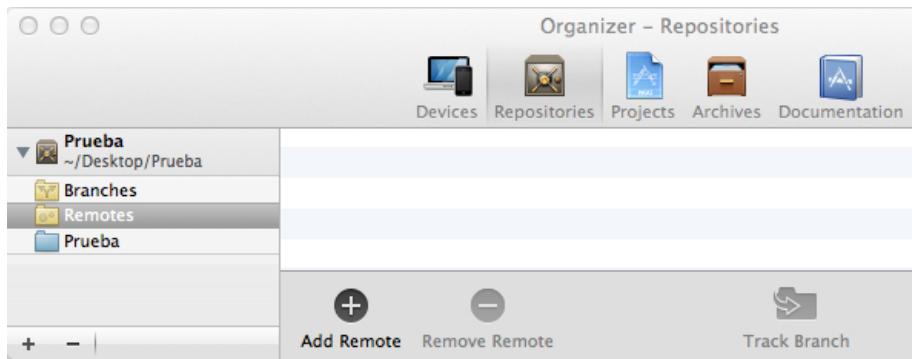
Creación de un repositorio BitBucket

- Una vez hecho esto, veremos el repositorio ya creado, en cuya ficha podremos encontrar la ruta que nos dará acceso a él.

The screenshot shows the BitBucket repository details page for 'Prueba'. At the top, there's a navigation bar with tabs for 'Overview', 'Downloads (0)', 'Pull requests (0)', 'Source', 'Commits', and 'Admin'. The 'Overview' tab is active. Below the tabs, the repository owner is listed as 'malozano / Prueba'. Underneath, there's a section for cloning the repository with the text 'Clone this repository (size: 580 bytes): [HTTPS](https://malozano@bitbucket.org/malozano/prueba.git) / [SSH](ssh://git@bitbucket.org/malozano/prueba.git) / [SourceTree](#)' and the command '\$ git clone https://malozano@bitbucket.org/malozano/prueba.git'.

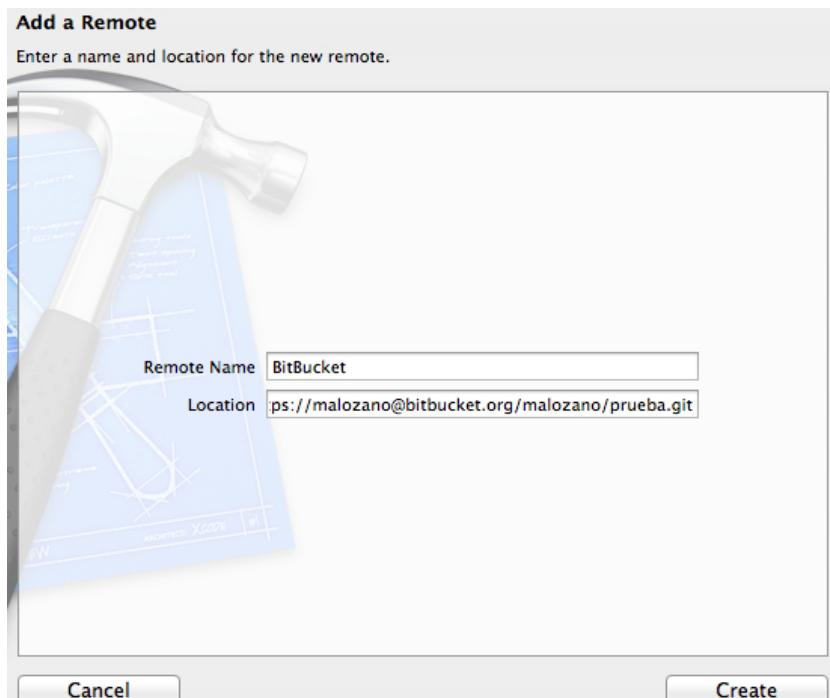
Ficha del repositorio en BitBucket

A continuación volvemos a Xcode, donde tenemos un proyecto ya creado con un repositorio Git local. Para vincular este repositorio local con el repositorio remoto que acabamos de crear, deberemos ir a la pestaña *Repositories* de la ventana del *Organizer*, donde veremos el repositorio local de nuestro proyecto.



Repositorio en Organizer

Dentro de este repositorio local, veremos una carpeta *Remotes*. Entraremos en ella, y pulsaremos el botón *Add Remotes* de la parte inferior de la pantalla. Ahora nos pedirá un nombre para el repositorio remoto y la ruta en la que se puede localizar. Utilizaremos aquí como ruta la ruta de tipo HTTPS que vimos en la ficha del repositorio de BitBucket:



Añadir repositorio remoto

Una vez añadido el repositorio remoto, deberemos introducir el login y password que nos den acceso a él. Con esto ya podemos volver al proyecto en Xcode, y enviar los cambios al servidor. En primer lugar deberemos hacer un *Commit* de los cambios en el repositorio local, si los hubiera (*File > Source Control > Commit...*). Tras esto, ya podremos subir estos cambios al repositorio remoto (*Push*). Para ello, seleccionamos la opción *File > Source Control > Push...*, tras lo cual nos pedirá que indiquemos el repositorio remoto en el que subir los cambios. Seleccionaremos el repositorio que acabamos de configurar.



Push en repositorio remoto

Tras realizar esta operación en la web de BitBucket podremos ver el código que hemos subido al repositorio:

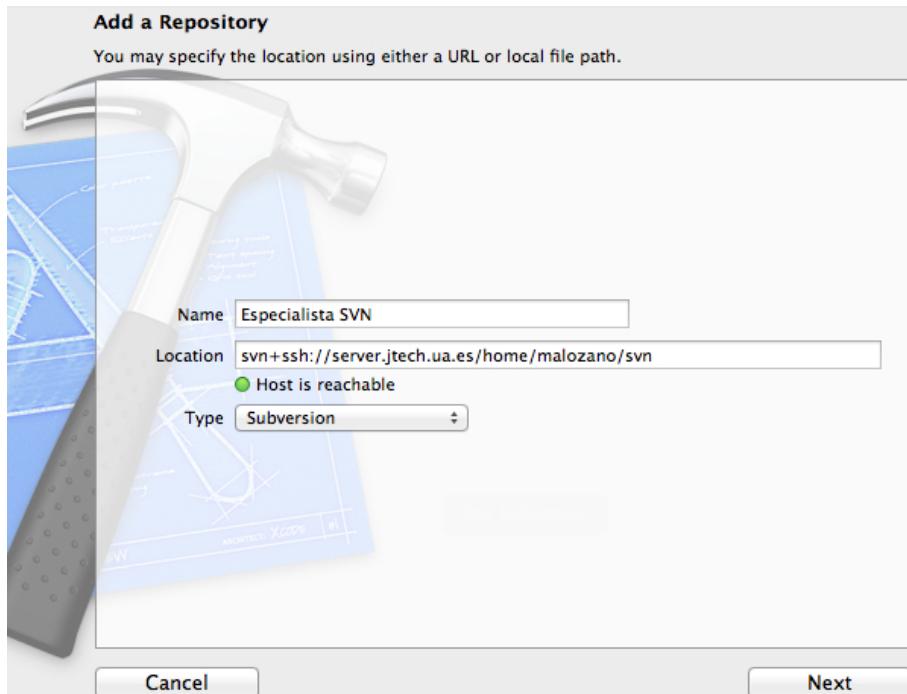
Código en BitBucket

1.9.2. Repositorios SVN

Vamos a ver la forma de acceder a un repositorio SVN remoto. Si queremos añadir un nuevo repositorio de forma manual deberemos pulsar el botón (+) que hay en la esquina inferior izquierda de la pantalla de repositorios de *Organizer*. Una vez pulsado el botón (+) y seleccionada la opción *Add Repository ...*, nos aparecerá la siguiente ventana donde introducir los datos de nuestro repositorio:

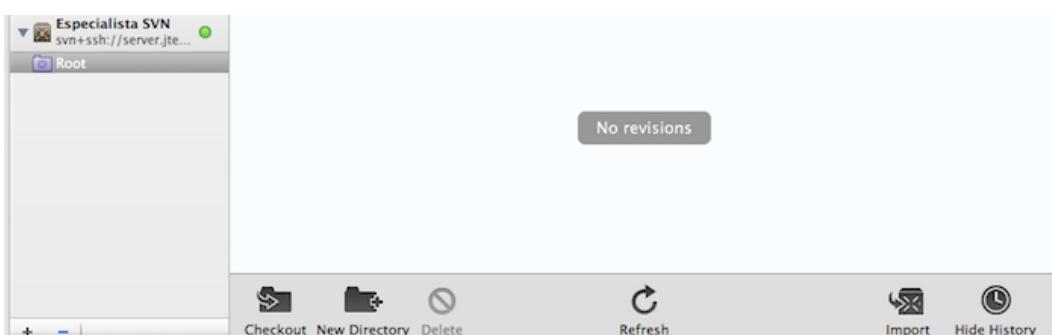
Le daremos un nombre y pondremos la ruta en la que se encuentra. Tras esto, pulsamos *Next*, y nos pedirá el login y password para acceder a dicho repositorio. Una vez

introducidos, nos llevará a una segunda pantalla donde nos pedirá las rutas de `trunk`, `branches`, y `tags` en el repositorio. Si el repositorio está todavía vacío, dejaremos estos campos vacíos y pulsaremos *Add* para añadir el nuevo repositorio.



Configurar un nuevo repositorio SVN

Una vez creado, lo veremos en el lateral izquierdo con un elemento *Root* que corresponderá a la raíz del repositorio. Si seleccionamos esta ruta raíz, en la zona central de la pantalla veremos su contenido y podremos crear directorios o importar ficheros mediante los botones de la barra inferior:



Acceso al repositorio SVN

Aquí podemos crear el *layout* del repositorio. Para repositorios SVN se recomienda que en el raíz de nuestro proyecto tengamos tres subdirectorios: `trunk`, `branches`, y `tags`. Podemos crearlos con el botón *New Directory*.

Una vez creados los directorios, pinchamos sobre el ítem con el nombre de nuestro repositorio en la barra de la izquierda e indicamos la localización de los directorios que acabamos de crear. Si ponemos bien las rutas junto a ellas aparecerá una luz verde:



Configuración del layout del repositorio

Además de *Root*, ahora veremos en nuestro repositorio también las carpetas de los elementos que acabamos de crear. El código en desarrollo se guardará en *trunk*, por lo que es ahí donde deberemos subir nuestro proyecto.

Por desgracia, no hay ninguna forma de compartir un proyecto de forma automática desde el entorno Xcode, como si que ocurre con Eclipse. De hecho, las opciones para trabajar con repositorios SCM en Xcode son muy limitadas y en muchas ocasiones en la documentación oficial nos indican que debemos trabajar desde línea de comando. Una de las situaciones en las que debe hacerse así es para subir nuestro proyecto por primera vez.

Vamos a ver una forma alternativa de hacerlo utilizando el *Organizer*, en lugar de línea de comando.

- Lo primero que necesitaremos es tener un proyecto creado con Xcode en el disco (cerraremos la ventana de Xcode del proyecto, ya que no vamos a seguir trabajando con esa copia).
- En el *Organizer*, entramos en el directorio *trunk* de nuestro repositorio y aquí seleccionamos *Import* en la barra inferior.
- Nos abrirá un explorador de ficheros para seleccionar los ficheros que queremos subir al repositorio, seleccionaremos el directorio raíz de nuestro proyecto (el directorio que contiene el fichero *xcodeproj*).
- Con esto ya tenemos el proyecto en nuestro repositorio SVN, pero si ahora abrimos el proyecto del directorio de nuestro disco local en el que está guardado, Xcode no reconocerá la conexión con el repositorio, ya que no es lo que se conoce como una *working copy* (no guarda la configuración de conexión con el repositorio SVN).
- Para obtener una *working copy* deberemos hacer un *checkout* del proyecto. Para ello desde el *Organizer*, seleccionamos la carpeta *Trunk* de nuestro repositorio, dentro de ella nuestro proyecto, y pulsamos el botón *Checkout* de la barra inferior. Nos pedirá un lugar del sistema de archivos donde guardar el proyecto, y una vez descargado nos preguntará si queremos abrirlo con Xcode.

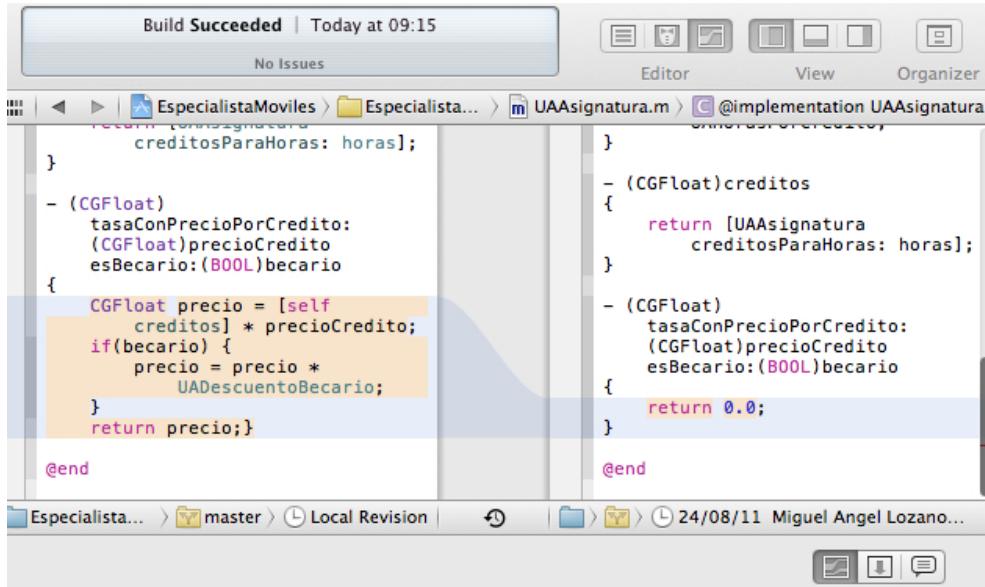
En los elementos del repositorio ahora veremos una carpeta azul con el nombre del proyecto descargado. Esta carpeta azul simboliza una *working copy* del proyecto. Al tener esta *working copy* reconocida, ya podremos realizar desde el mismo entorno Xcode operaciones de SVN en dicho proyecto.



Cada vez que hagamos un cambio en un fichero, en el navegador de Xcode aparecerá una **M** a su lado indicando que hay modificaciones pendientes de ser enviadas, y de la misma forma, cuando añadamos un fichero nuevo, junto a él aparecerá una **A** para indicar que está pendiente de ser añadido al repositorio. Para enviar estos cambios al repositorio, desde Xcode seleccionaremos *File > Source Control > Commit*.

Si ya tuviesemos una *working copy* válida de un proyecto en el disco (es decir, una copia que contenga los directorios `.svn` con la configuración de acceso al repositorio), podemos añadirla al *Organizer* directamente desde la vista de repositorios, pulsando el botón (+) de la esquina inferior izquierda y seleccionando *Add Working Copy....* Nos pedirá la ruta local en la que tenemos guardada la *working copy* del proyecto.

A parte de las opciones para el control de versiones que encontramos al pulsar con el botón derecho sobre nuestro proyecto en el apartado *Source Control*, en Xcode también tenemos el editor de versiones. Se trata de una vista del editor que nos permite comparar diferentes versiones de un mismo fichero, y volver atrás si fuese necesario. Para abrir este editor utilizaremos los botones de la esquina superior derecha de la interfaz, concretamente el tercer botón del grupo *Editor*. En el editor de versiones podremos comparar la revisión local actual de cada fichero con cualquiera de las revisiones anteriores.

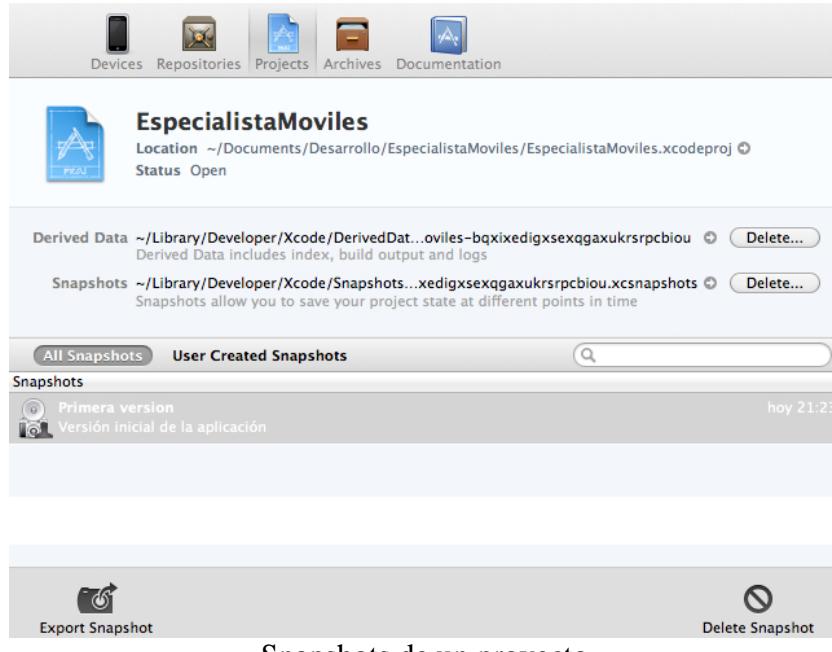


1.10. Snapshots

Una forma más rudimentaria de guardar copias de seguridad de los proyectos es la generación de *snapshots*. Un *snapshot* no es más que la copia de nuestro proyecto completo en un momento dado. Xcode se encarga de archivar y organizar estas copias.

Para crear un *snapshot* debemos seleccionar en Xcode *File > Create Snapshot ...*, para el cual nos pedirá un nombre y una descripción.

En la vista *Projects* del *Organizer* veremos para cada proyecto la lista de *snapshots* creados.



Desde aquí podremos exportar un *snapshot* para guardar el proyecto en el lugar del disco que especifiquemos tal como estaba en el momento en el que se tomó la instantánea, y así poder abrirlo con Xcode. En esta pantalla también podemos limpiar los directorios de trabajo del proyecto y borrar los *snapshots*.

Desde Xcode, también se puede restaurar un *snapshot* con la opción *File > Restore snapshot....* Tendremos que seleccionar el *snapshot* que queremos restaurar.

1.11. Ejecución y firma

Como hemos comentado anteriormente, para probar las aplicaciones en dispositivos reales necesitaremos un certificado con el que firmarlas, y para obtener dicho certificado tendremos que ser miembros de pago del *iOS Developer Program*, o bien pertenecer al *iOS University Program*. Para acceder a este segundo programa, se deberá contar con una cuenta Apple, y solicitar al profesor responsable una invitación para acceder al programa con dicha cuenta. El programa es gratuito, pero tiene la limitación de que no podremos publicar las aplicaciones en la App Store, sólo probarlas en nuestros propios dispositivos.

Vamos a ver a continuación cómo obtener dichos certificados y ejecutar las aplicaciones en dispositivos reales.

1.11.1. Creación del par de claves

El primer paso para la obtención del certificado es crear nuestras claves privada y pública. Para ello entraremos en el portal de desarrolladores de Apple:

<http://developer.apple.com>

Accedemos con nuestro usuario, que debe estar dado de alta en alguno de los programas comentados anteriormente.

The screenshot shows the Apple Developer website's iOS Dev Center. At the top, there are links for Technologies, Resources, Programs, Support, and Member Center, along with a search bar. Below the header, it says "iOS Dev Center" and shows "Hi, Miguel Angel Lozano Ortega" with links to "My Profile" and "Log out". On the left, there's a sidebar with "iOS SDK 4.3" (beta), "Resources for iOS 4.3" (Downloads and Getting Started Videos), and "Featured Content" (New Subscription Service for iOS Apps and Getting Ready for iOS 4.3). On the right, there's a box for the "iOS Developer Program" with links to the "iOS Provisioning Portal", "Apple Developer Forums", and "Developer Support Center". The main content area is titled "Developer Center".

Desde este portal, accedemos al *iOS Provisioning Portal*.

The screenshot shows the "Provisioning Portal : Universidad de Alicante (Dpto. de Ciencia de la Computacion e I.A.)". The left sidebar has links for Home, Certificates, Devices, App IDs, and Provisioning. The main content area is titled "Welcome to the iOS Provisioning Portal" and says: "The iOS Provisioning Portal is designed to take you through the necessary steps to test your applications on iOS devices and prepare them for distribution." It features a callout box with an exclamation mark icon that says: "Visit the Member Center for Team, Account, and Program info". It explains that the new Member Center is now the destination for: "Sending invitations to join your development team and editing existing development team members.", "Requesting or purchasing Technical Support.", and "Viewing account information, such as your Team ID, profile, and Program details.". A link "Visit the Member Center now" is provided. Below this, there's another box with an iPhone icon and the text: "Get your application on an iOS with the Development Provisioning Assistant". It says: "As a Program Admin, you can use the Development Provisioning Assistant to create and install a Provisioning Profile and iOS Development Certificate needed to build and install applications you're developing for iOS devices." A "Launch Assistant" button is shown. The overall title at the bottom is "Provisioning Portal".

Desde aquí podemos gestionar nuestros certificados y los dispositivos registrados para poder utilizarlos en el desarrollo de aplicaciones. Vemos también que tenemos disponible

un asistente para crear los certificados por primera vez. El asistente se nos irá dando instrucciones detalladas y se encargará de configurar todo lo necesario para la obtención de los certificados y el registro de los dispositivos. Sin embargo, con Xcode 4 el registro de los dispositivos se puede hacer automáticamente desde el *Organizer*, así que nos resultará más sencillo si en lugar del asistente simplemente utilizamos el *iOS Provisioning Portal* para obtener el certificado del desarrollador, y dejamos el resto de tareas a *Organizer*.

The screenshot shows the 'Certificates' section of the iOS Provisioning Portal. The left sidebar has links for Home, Certificates (which is selected), Devices, App IDs, and Provisioning. The main area has tabs for Development, History, and How To. Under 'Development', it says 'Current Development Certificates'. It shows a table with columns: Name, Provisioning Profiles, Expiration Date, Status, and Action. A message in the table says 'You currently do not have a valid certificate' and has a 'Request Certificate' button. Below the table is a note: '*If you do not have the WWDR Intermediate certificate installed, click here to download now.'

Gestión de certificados de desarrollador

Entraremos por lo tanto en *Certificates*, y dentro de la pestaña *Development* veremos los certificados con los que contamos actualmente (si es la primera vez que entramos no tendremos ninguno), y nos permitirá solicitarlos o gestionarlos. Antes de solicitar un nuevo certificado, descargaremos desde esta página el certificado WWDR, y haremos doble *click* sobre él para instalarlo en nuestros llaveros (*keychain*). Este certificado es necesario porque es el que valida los certificados de desarrollador que emite Apple. Sin él, el certificado que generemos no sería de confianza.

Una vez descargado e instalado dicho certificado intermedio, pulsaremos sobre el botón *Request Certificate* para comenzar con la solicitud de un nuevo certificado de desarrollador.

Nos aparecerán las instrucciones detalladas para solicitar el certificado. Deberemos utilizar la herramienta *Acceso a Llaveros (Keychain Access)* para realizar dicha solicitud, tal como explican las instrucciones. Abrimos la herramienta y accedemos a la opción del menú *Acceso a Llaveros > Asistente para Certificados > Solicitar un certificado de una autoridad de certificación*

Información del certificado

Introduzca información para el certificado que está solicitando.
Haga clic en continuar para solicitar un certificado de la CA.

Dirección de correo del usuario:	<input type="text" value="malozano@ua.es"/>
Nombre común:	<input type="text" value="Miguel Angel Lozano"/>
Dirección de correo de la CA:	<input type="text"/>
La palabra clave:	<input type="radio"/> Se envía por correo electrónico a la CA <input checked="" type="radio"/> Se guarda en el disco <input type="checkbox"/> Permitirme especificar la información del par de llaves

Continuar

Solicitud de un certificado de desarrollador

Aquí tendremos que poner nuestro *e-mail*, nuestro nombre, e indicar que se guarde en el disco. Una vez finalizado, pulsamos *Continuar* y nos guardará un fichero *.certSigningRequest* en la ubicación que seleccionemos.

Con esto habremos generado una clave privada, almacenada en los llaveros, y una clave pública que se incluye en el fichero de solicitud de certificado. Volvemos ahora al *iOS Provisioning Portal*, y bajo las instrucciones para la solicitud del certificado vemos un campo para enviar el fichero. Enviaremos a través de dicho campo el fichero de solicitud (*.certSigningRequest*) generado, y en la página *Certificates > Development* aparecerá nuestro certificado como pendiente. Cuando Apple emita el certificado podremos descargarlo a través de esta misma página (podemos recargarla pasados unos segundos hasta que el certificado esté disponible).

[Development](#) [History](#) [How To](#)

Current Development Certificates

 **Your Certificate**

Name	Provisioning Profiles	Expiration Date	Status	Action
Miguel Angel Lozano Ortega		Aug 23, 2012	Issued	Download Revoke

*If you do not have the WWDR intermediate certificate installed, [click here to download now.](#)

Certificado listo para descargar

Una vez esté disponible, pulsaremos el botón *Download* para descargarlo y haremos doble *click* sobre él para instalarlo en nuestros llaveros. Una vez hecho esto, en *Acceso a*

Llaveros > Inicio de sesión > Mis certificados deberemos ver nuestro certificado con la clave privada asociada.



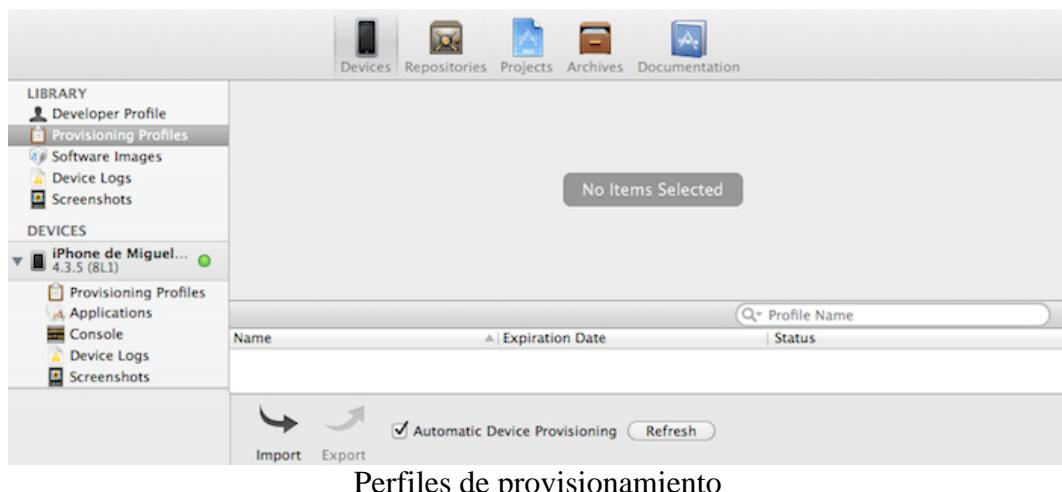
Importante

El certificado generado no podrá utilizarse en otros ordenadores para firmar la aplicación, ya que la clave privada está incluida en nuestros llaveros, pero no en el certificado, y será necesaria para poder firmar. Si queremos trabajar en otro ordenador tendremos que exportar la clave privada desde *Acceso a Llaveros*.

1.11.2. Perfil de provisionamiento

Una vez contamos con nuestro certificado de desarrollador instalado y con su clave privada asociada en nuestro llavero, podemos crear un perfil de provisionamiento para poder probar nuestras aplicaciones en nuestros dispositivos.

Para ello abriremos el *Organizer* e iremos a la sección *Devices*. Aquí entraremos en *Developer Profiles* y comprobaremos que ha reconocido correctamente nuestro perfil de desarrollador tras instalar el certificado. Ahora lo que necesitamos son perfiles de provisionamiento para poder instalar la aplicación en dispositivos concretos. Si entramos en *Provisioning Profiles* veremos que no tenemos ninguno por el momento, y nos aseguraremos que en la parte inferior tengamos marcado *Automatic Device Provisioning*.



Perfiles de provisionamiento

Ahora conectaremos mediante el cable USB el dispositivo que queramos registrar y veremos que aparece en la barra de la izquierda. Pulsamos sobre el nombre del dispositivo en dicha barra y abrirá una pantalla con toda su información, entre ella, la versión de iOS y el identificador que se utilizará para registrarla en el *iOS Provisioning*

Portal.



Gestión de los dispositivos

Vemos que no tiene ningún perfil de provisionamiento disponible, pero abajo vemos un botón *Add to Portal* que nos permite registrar automáticamente nuestro dispositivo para desarrollo. Al pulsarlo nos pedirá nuestro *login* y *password* para acceder al portal de desarrolladores, y automáticamente se encargará de registrar el dispositivo, crear los perfiles de provisionamiento necesarios, descargarlos e instalarlos en Xcode.

Nota

Si estamos usando el *iOS University Program* al final del proceso, cuando esté obteniendo el perfil de distribución, nos saldrá un mensaje de error indicando que nuestro programa no permite realizar dicha acción. Esto es normal, ya que el perfil universitario no permite distribuir aplicaciones, por lo que la generación del perfil de distribución deberá fallar. Pulsaremos *Aceptar* y el resto del proceso se realizará correctamente.

Si ahora volvemos al *iOS Developer Portal* veremos que nuestro dispositivo se encuentra registrado ahí (sección *Devices*), y que contamos con un perfil de provisionamiento (sección *Provisioning*).

Ahora podemos volver a Xcode y ejecutar nuestra aplicación en un dispositivo real. Para ello seleccionamos nuestro dispositivo en el cuadro junto a los botones de ejecución, y pulsamos el botón *Run*.



Ejecutar en un dispositivo real

2. Ejercicios de Xcode

2.1. Creación de un proyecto con Xcode

- a) Vamos a crear un nuevo proyecto de tipo *Master-Detail Application*. El nombre de producto será *Filmoteca* y nuestra organización *es.ua.jtech*. Utilizaremos como prefijo para todas las clases *UA*, y sólo destinaremos la aplicación al iPhone por el momento. También desmarcaremos todas las casillas adicionales, ya que no vamos a utilizar por el momento ninguna de esas características (Core Data, ARC, pruebas de unidad, storyboards).
- b) ¿Qué versión de iOS requiere la aplicación? Modifica la configuración necesaria para que funcione en dispositivos con iOS 4.0.
- c) Ejecuta la aplicación en el simulador de iPhone. Comprueba que funciona correctamente.

2.2. Iconos y recursos

Vamos a añadir ahora un ícono al proyecto, y una imagen a mostrar durante la carga de la aplicación. Estas imágenes se proporcionan en el fichero de plantillas de la sesión. Se pide:

- a) Añadir la imagen `icono.png` como ícono de la aplicación. Para hacer esto podemos seleccionar en el navegador el nodo principal del proyecto, y en la pestaña *Summary* del *target* principal veremos una serie de huecos para indicar los íconos. Bastará con arrastrar la imagen al hueco correspondiente.
- b) Añade la imagen `icono@2x.png` como ícono para los dispositivos con pantalla retina. Puedes hacer que el simulador tenga pantalla retina, mediante la opción *Hardware > Dispositivo > iPhone (Retina)*. Comprueba que la imagen se ve con la definición adecuada a esta versión del simulador.
- c) Añade las imágenes `Default.png` y `Default@2x.png` como imágenes a mostrar durante el tiempo de carga de la aplicación (*Launch Images*). Comprueba que aparecen correctamente.

2.3. Localización

Vamos ahora a localizar los textos de nuestra aplicación. Se pide:

- a) Añade a la aplicación las localizaciones a inglés y español.
- b) Haz que el nombre que se muestra bajo el ícono de la aplicación en el iPhone cambie

según el idioma del dispositivo (`Filmoteca`, `FilmLibrary`). Puedes probar que los distintos idiomas se muestran correctamente cambiando el idioma del simulador (*Settings > General > International > Language*).

- c) No queremos localizar los ficheros `.xib`. Elimina la localización de dichos ficheros.
- d) Crea un fichero `Localizable.strings`, y en él una cadena para el título de la aplicación que tenga como identificador `"AppName"`, y como valor `Filmoteca` o `FilmLibrary`, según el idioma del dispositivo.

3. Introducción a Objective-C

Prácticamente toda la programación de aplicaciones iOS se realizará en lenguaje Objective-C, utilizando la API Cocoa Touch. Este lenguaje es una extensión de C, por lo que podremos utilizar en cualquier lugar código C estándar, aunque normalmente utilizaremos los elementos equivalentes definidos en Objective-C para así mantener una coherencia con la API Cocoa, que está definida en dicho lenguaje.

Vamos a ver los elementos básicos que aporta Objective-C sobre los elementos del lenguaje con los que contamos en lenguaje C estándar. También es posible combinar Objective-C y C++, dando lugar a Objective-C++, aunque esto será menos común.

3.1. Tipos de datos

3.1.1. Tipos de datos básicos

A parte de los tipos de datos básicos que conocemos de C (char, short, int, long, float, double, unsigned int, etc), en Cocoa se definen algunos tipos numéricos equivalentes a ellos que encontraremos frecuentemente en dicha API:

- `NSInteger` (int o long)
- `NSUInteger` (unsigned int o unsigned long)
- `CGFloat` (float o double)

Podemos asignar sin problemas estos tipos de Cocoa a sus tipos C estándar equivalentes, y al contrario.

Contamos también con el tipo booleano definido como `BOOL`, y que puede tomar como valores las constantes YES (1) y NO (0):

```
BOOL activo = YES;
```

3.1.2. Enumeraciones

Las enumeraciones resultan útiles cuando una variable puede tomar su valor de un conjunto limitado de posibles opciones. Dentro de la API de Cocoa es habitual encontrar enumeraciones, y se definen de la misma forma que en C estándar:

```
typedef enum {
    UATipoAsignaturaOptativa,
    UATipoAsignaturaObligatoria,
    UATipoAsignaturaTroncal
} UATipoAsignatura;
```

A cada elemento de la enumeración se le asigna un valor entero, empezando desde 0, y de forma incremental siguiendo el orden en el que está definida la enumeración. Podemos

también especificar de forma manual el número asignado a cada elemento.

```
typedef enum {
    UATipoAsignaturaOptativa = 0,
    UATipoAsignaturaObligatoria = 1,
    UATipoAsignaturaTroncal = 2
} UATipoAsignatura;
```

3.1.3. Estructuras

También encontramos y utilizamos estructuras de C estándar dentro de la API de Cocoa.

```
struct CGPoint {
    CGFloat x;
    CGFloat y;
};
typedef struct CGPoint CGPoint;
```

Muchas veces encontramos librerías de funciones para inicializarlas o realizar operaciones con ellas.

```
CGPoint punto = CGPointMake(x,y);
```

3.1.4. Cadenas

En C normalmente definimos una cadena entre comillas, por ejemplo "cadena". Con esto estamos definiendo un *array* de caracteres terminado en `null`. Esto es lo que se conoce como cadena C, pero en Objective-C normalmente no utilizaremos dicha representación. Las cadenas en Objective-C se representarán mediante la clase `NSString`, y los literales de este tipo en el código se definirán anteponiendo el símbolo @ a la cadena:

```
NSString* cadena = @"cadena";
```

Nota

Todas las variables para acceder a objetos de Objective-C tendrán que definirse como punteros, y por lo tanto en la declaración de la variable tendremos que poner el símbolo *. Los únicos tipos de datos que no se definirán como punteros serán los tipos básicos y las estructuras.

Más adelante veremos las operaciones que podemos realizar con la clase `NSString`, entre ellas el convertir una cadena C estándar a un `NSString`.

3.1.5. Objetos

Las cadenas en Objective-C son una clase concreta de objetos, a diferencia de las cadenas C estándar, pero las hemos visto como un caso especial por la forma en la que podemos definir en el código literales de ese tipo.

En la API de Objective-C encontramos una extensa librería de clases, y normalmente deberemos también crear clases propias en nuestras aplicaciones. Para referenciar una

instancia de una clase siempre lo haremos mediante un puntero.

```
MiClase* objeto;
```

Sin embargo, como veremos más adelante, la forma de trabajar con dicho puntero diferirá mucho de la forma en la que se hacía en C, ya que Objective-C se encarga de ocultar toda esa complejidad subyacente y nos ofrece una forma sencilla de manipular los objetos, más parecida a la forma con la que trabajamos con la API de Java.

No obstante, podemos utilizar punteros al estilo de C. Por ejemplo, podemos crearnos punteros de tipos básicos o de estructuras, pero lo habitual será trabajar con objetos de Objective-C.

Otra forma de hacer referencia a un objeto de Objective-C es mediante el tipo `id`. Cuando tengamos una variable de este tipo podremos utilizarla para referenciar cualquier objeto, independientemente de la clase a la que pertenezca. El compilador no realizará ningún control sobre los métodos a los que llamemos, por lo que deberemos llevar cuidado al utilizarlo para no obtener ningún error en tiempo de ejecución.

```
MiClase* mc = // Inicializa MiClase;
MiOtraClase* moc = // Inicializa MiOtraClase;
...
id referencia = nil;
referencia = mc;
referencia = moc;
```

Como vemos, `id` puede referenciar instancias de dos clases distintas sin que exista ninguna relación entre ellas. Hay que destacar también que las referencias a objetos con `id` no llevan el símbolo `*`.

También observamos que para indicar un puntero de objeto a nulo utilizamos `nil`. También contamos con `NULL`, y ambos se resuelven de la misma forma, pero conceptualmente se aplican a casos distintos. En caso de `nil`, nos referimos a un puntero nulo a un objeto de Objective-C, mientras que utilizamos `NULL` para otros tipos de punteros.

3.2. Directivas

También podemos incluir en el código una serie de directivas de preprocesamiento que resultarán de utilidad y que encontraremos habitualmente en aplicaciones iOS.

3.2.1. La directiva `#import`

Con esta directiva podemos importar ficheros de cabecera de librerías que utilicemos en nuestro código. Se diferencia de `#include` en que con `#import` se evita que un mismo fichero sea incluido más de una vez cuando encontremos inclusiones recursivas.

Encontramos dos versiones de esta directiva: `#import<...>` e `#import "..."`. La primera de ellas buscará los ficheros en la ruta de inclusión del compilador, por lo que utilizaremos esta forma cuando estemos incluyendo las librerías de Cocoa. Con la segunda, se incluirá también nuestro propio directorio de fuentes, por lo que se utilizará para importar el código de nuestra propia aplicación.

3.2.2. La directiva `#define`

Este directiva toma un nombre (símbolo) y un valor, y sustituye en el código todas las ocurrencias de dicho nombre por el valor indicado antes de realizar la compilación. Como valor podremos poner cualquier expresión, ya que la sustitución se hace como preprocesamiento antes de compilar.

También podemos definir símbolos mediante parámetros del compilador. Por ejemplo, si nos fijamos en las *Build Settings* del proyecto, en el apartado *Preprocessing* vemos que para la configuración *Debug* se le pasa un símbolo `DEBUG` con valor 1.

Tenemos también las directivas `#ifdef` (y `#ifndef`) que nos permiten incluir un bloque de código en la compilación sólo si un determinado símbolo está definido (o no). Por ejemplo, podemos hacer que sólo se escriban logs si estamos en la configuración *Debug* de la siguiente forma:

```
#ifdef DEBUG  
    NSLog(@"%@", @"Texto del log");  
#endif
```

Los nombres de las constantes (para ser más exactos macros) definidas de esta forma normalmente se utilizan letras mayúsculas separando las distintas palabras con el carácter subrayado `'_'` (`UPPER_CASE_UNDERSCORE`).

3.2.3. La directiva `#pragma mark`

Se trata de una directiva muy utilizada en los ficheros de fuentes de Objective-C, ya que es reconocida y utilizada por Xcode para organizar nuestro código en secciones. Con dicha directiva marcamos el principio de cada sección de código y le damos un nombre. Con la barra de navegación de Xcode podemos saltar directamente a cualquier de las secciones:

```
#pragma mark Constructores  
  
// Código de los constructores  
  
#pragma mark Eventos del ciclo de vida  
  
// Código de los manejadores de eventos  
  
#pragma mark Fuente de datos  
  
// Métodos para la obtención de datos  
  
#pragma mark Gestión de la memoria
```

```
// Código de gestión de memoria
```

3.2.4. Modificadores

Tenemos disponibles también modificadores que podemos utilizar en la declaración de las variables.

3.2.4.1. Modificador const

Indica que el valor de una variable no va a poder ser modificado. Se le debe asignar un valor en la declaración, y este valor no podrá cambiar posteriormente. Se diferencia de `#define` en que en este caso tenemos una variable en tiempo de compilación, y no una sustitución en preprocessamiento. En general, no es recomendable utilizar `#define` para definir las constantes. En su lugar utilizaremos normalmente `enum` o `const`.

Hay que llevar cuidado con el lugar en el que se declara `const` cuando se trate de punteros. Siempre afecta al elemento que tenga inmediatamente a la izquierda, excepto en el caso en el que esté al principio, que afectará al elemento de la derecha:

```
// Puntero variable a objeto NSString constante (MAL)
const NSString * UATitulo = @"Menu";

// Equivalente al anterior (MAL)
NSString const * UATitulo = @"Menu";

// Puntero constante a objeto NSString (BIEN)
NSString * const UATitulo = @"Menu";
```

En los dos primeros casos, estamos definiendo un puntero a un objeto de tipo `const NSString`. Por lo tanto, nos dará un error si intentamos utilizarlo en cualquier lugar en el que necesitemos tener un puntero a `NSString`, ya que el compilador los considera tipos distintos. Además, no es necesario hacer que `NSString` sea constante. Para ello en la API de Cocoa veremos que existen versiones mutables e inmutables de un gran número de objetos. Bastará con utilizar una cadena inmutable.

Las constantes se escribirán en *UpperCamelCase*, utilizando el prefijo de nuestra librería en caso de que sean globales.

3.2.4.2. Modificador static

Nos permite indicar que una variable se instancie sólo una vez. Por ejemplo en el siguiente código:

```
static NSString *cadena = @"Hola";
NSLog(cadena);
cadena = @"Adios";
```

La primera vez que se ejecute, la variable `cadena` se instanciará y se inicializará con el valor `@"Hola"`, que será lo que se escriba como *log*, y tras ello modificaremos su valor a

`@"Adios".` En las sucesivas ejecuciones, como la variable ya estaba instanciada, no se volverá a instanciar ni a inicializar, por lo que al ejecutar el código simplemente escribirá Adios.

Como veremos más adelante, este modificador será de gran utilidad para implementar el patrón *singleton*.

El comportamiento de `static` difiere según si la variable sobre la que se aplica tiene ámbito local o global. En el ámbito local, tal como hemos visto, nos permite tener una variable local que sólo se instancia una vez a lo largo de todas las llamadas que se hagan al método. En caso de aplicarlo a una variable global, indica que dicha variable sólo será accesible desde dentro del fichero en el que esté definida. Esto resultará de utilidad por ejemplo para definir constantes privadas que sólo vayan a utilizarse dentro de un determinado fichero .m. Al comienzo de dicho ficheros podríamos declararlas de la siguiente forma:

```
static NSString * const titulo = @"Menu";
```

Si no incluimos `static`, si en otro fichero se definiere otro símbolo global con el mismo nombre, obtendríamos un error en la fase de *linkado*, ya que existirían dos símbolos con el mismo nombre dentro del mismo ámbito.

3.2.4.3. Modificador extern

Si en el ejemplo anterior quisiéramos definir una constante global, no bastaría con declarar la constante sin el modificador `static` en el fichero .m:

```
NSString * const titulo = @"Menu";
```

Si sólo hacemos esto, aunque el símbolo sea accesible en el ámbito global, el compilador no va a ser capaz de encontrarlo ya que no hemos puesto ninguna declaración en los ficheros de cabecera incluidos. Por lo tanto, además de la definición anterior, tendremos que declarar dicha constante en algún fichero de cabecera con el modificador `extern`, para que el compilador sepa que dicho símbolo existe en el ámbito global.

Para concluir, listamos los tres posibles ámbitos en los que podemos definir cada símbolo:

- **Global:** Se declaran fuera de cualquier método para que el símbolo se guarde de forma global. Para que el compilador sepa que dicho símbolo existe, se deben declarar en los ficheros de cabecera con `extern`. Sólo se instancian una vez.
- **Fichero:** Se declaran fuera de cualquier método con modificador `static`. Sólo se podrá acceder a ella desde dentro del fichero en el que se ha definido, por lo que no deberemos declararlas en ningún fichero de cabecera que vaya a ser importado por otras unidades. Sólo se instancian una vez.
- **Local:** Se declaran dentro de un bloque de código, como puede ser una función, un método, o cualquier estructura que contengan los anteriores, y sólo será accesible dentro de dicho bloque. Por defecto se instanciarán cada vez que entremos en dicho

bloque, excepto si se declaran con el modificador `static`, caso en el que se instanciarán y se inicializarán sólo la primera vez que se entre.

3.3. Paso de mensajes

Como hemos comentado anteriormente, Objective-C es una extensión de C para hacerlo orientado a objetos, como es también el caso de C++. Una de las mayores diferencias entre ambos radica en la forma en la se ejecutan los métodos de los objetos. En Objective-C los métodos siempre se ejecutan de forma dinámica, es decir, el método a ejecutar no se determina en tiempo de compilación, sino en tiempo de ejecución. Por eso hablamos de *paso de mensajes*, en lugar de *invocar* un método. La forma en la que se pasan los mensajes también resulta bastante peculiar y probablemente es lo que primero nos llame la atención cuando veamos código Objective-C:

```
NSString* cadena = @"cadena-de-prueba";
NSUInteger tam = [cadena length];
```

Podemos observar que para pasar un mensaje a un objeto, ponemos entre corchetes [...] la referencia al objeto, y a continuación el nombre del método que queramos ejecutar.

Los métodos pueden tomar parámetros de entrada. En este caso cada parámetro tendrá un nombre, y tras poner el nombre del parámetro pondremos : seguido de el valor que queramos pasarle:

```
NSString* result = [cadena stringByReplacingOccurrencesOfString: @"-"
                                                       withString: @" "];
```

Podemos observar que estamos llamando al método `stringByReplacingOccurrencesOfString:withString:` de nuestro objeto de tipo `NSString`, para reemplazar los guiones por espacios.

Es importante remarcar que el nombre de un método comprende el de todos sus parámetros, por ejemplo, el método anterior se identificaría mediante `stringByReplacingOccurrencesOfString:withString:`. Esto es lo que se conoce como un *selector*, y nos permite identificar los mensajes que se le van a pasar a un objeto.

No podemos sobrecargar los métodos, es decir, no puede haber dos métodos que correspondan a un mismo *selector* pero con distinto tipo de parámetros. Sin embargo, si que podemos crear varias versiones de un método con distinto número o nombres de parámetros. Por ejemplo, también existe el método `stringByReplacingOccurrencesOfString:withString:options:range:` que añade dos parámetros adicionales al anterior.

Es posible llamar a métodos inexistentes sin que el compilador nos lo impida, como mucho obtendremos un *warning*:

```
NSString* cadena = @"cadena-de-prueba";
[cadena metodoInexistente]; // Produce warning, pero compila
```

En el caso anterior, como `NSString` no tiene definido ningún método que se llame `metodoInexistente`, el compilador nos dará un *warning*, pero la aplicación compilará, y tendremos un error en tiempo de ejecución. Concretamente saltará una excepción que nos indicará que se ha enviado un mensaje a un selector inexistente.

En el caso en que nuestra variables fuese de tipo `id`, ni siquiera obtendríamos ningún *warning en la compilación*. En este tipo de variables cualquier mensaje se considera válido:

```
id cadena = @"cadena-de-prueba";
[cadena metodoInexistente]; // Solo da error de ejecucion
```

Por este motivo es por el que hablamos de paso de mensajes en lugar de llamadas a métodos. Realmente lo que hace nuestro código es enviar un mensaje al objeto, sin saber si el método existe o no.

3.4. Clases y objetos

En Objective-C cada clase normalmente se encuentra separada en dos ficheros: la declaración de la interfaz en un fichero `.h`, y la implementación en un `.m`. A diferencia de Java, el nombre del fichero no tiene que coincidir con el nombre de la clase, y podemos tener varias clases definidas en un mismo fichero. Podremos utilizar cualquiera de ellas siempre que importemos el fichero `.h` correspondiente.

El criterio que se seguirá es el de agrupar en el mismo fichero aquellas clases, estructuras, funciones y elementos adicionales que estén muy relacionados entre sí, y dar al fichero el nombre de la clase principal que contiene. Por ejemplo, las clases `NSString` y `NSMutableString` se definen en el mismo fichero, de nombre `NSString.h(m)`. La segunda es una subclase de la primera, que añade algunos métodos para poder modificar la cadena. Si estamos creando un *framework*, también deberemos crear un único fichero `.h` que se encargue de importar todos los elementos de nuestro *framework*. Por ejemplo, si queremos utilizar el *framework Foundation* sólo necesitamos importar `Foundation/Foundation.h`, a pesar de que las clases de esta librería se declaran en diferentes ficheros de cabecera.

La forma más rápida de crear una nueva clase con Xcode es seleccionar *File > New > New File... > Cocoa Touch > Objective-C class*. Nos permitirá especificar la superclase, poniendo por defecto `NSObject`, que es la superclase en última instancia de todas las clases, como ocurre en Java con `Object`. Tras esto, tendremos que dar un nombre y una ubicación a nuestra clase, y guardará los ficheros `.h` y `.m` correspondientes.

3.4.1. Declaración de una clase

En el fichero `.h`, en la declaración de la interfaz vemos que se indica el nombre de la clase seguido de la superclase:

```
@interface MiClase : NSObject
@end
```

Podemos introducir entre llaves una serie de variables de instancia.

```
@interface UAAsignatura : NSObject {
    NSString *_nombre;
    NSString *_descripcion;
    NSUInteger _horas;
}
@end
```

Una cosa importante que debemos tener en cuenta es que las variables de instancia en Objective-C son **por defecto protegidas**. Podemos añadir los modificadores de acceso `@public`, `@protected`, y `@private`, pero no es recomendable declarar variables públicas. En su lugar, para poder acceder a ellas añadiremos métodos *accesores*. Lo más común será dejar el nivel de acceso por defecto.

```
@interface MiClase : NSObject {
    @public
        NSString *_varPublica;
        NSInteger _otraVarPublica;
    @protected
        NSString *_varProtegida;
        NSInteger _otraVarProtegida;
    @private
        NSString *_varPrivada;
        NSInteger _otraVarPrivada;
}
@end
```

Los nombres de las variables de instancia se escriben en *lowerCamelCase*, y su nombre debe resultar descriptivo, evitando abreviaturas, excepto las utilizadas de forma habitual (como `min` o `max`).

Los métodos se declaran dentro del bloque `@interface`, pero fuera de las llaves. La firma de cada método comienza por `+` o `-`, según sea un método de clase o de instancia respectivamente. Tras este indicador, se indica el tipo de datos que devuelve el método, y a continuación el nombre del método, sus parámetros y sus tipos:

```
@interface UAAsignatura : NSObject {
    NSString *_nombre;
    NSString *_descripcion;
    NSUInteger _horas;
}

+ (CGFloat)creditosParaHoras:(CGFloat)horas;
- (CGFloat)creditos;
- (CGFloat)tasaConPrecioPorCredito:(CGFloat)precioCredito
    esBecario:(BOOL)becario;

@end
```

En el ejemplo hemos definido tres métodos que para calcular los créditos de una asignatura y su precio. El primero de ellos nos dice a cuántos créditos corresponde un

número de horas dado como parámetro. Dado que no necesita acceder a las variables de ninguna instancia concreta lo hemos definido como método de clase, para así poderlo ejecutar simplemente a partir del nombre de la clase:

```
CGFloat cr = [UAAsignatura creditosParaHoras: 20];
```

Los otros métodos los tendremos que ejecutar a partir de una instancia concreta de la clase:

```
UAAsignatura *asignatura = // Instanciar clase  
CGFloat creditos = [asignatura creditos];  
CGFloat precio = [asignatura tasaConPrecioPorCredito: 60 esBecario: NO];
```

Espacios de nombres

En Objective-C no tenemos un espacio de nombres para nuestras clases como son los paquetes de Java, por lo que corremos el riesgo de que el nombre de una de nuestras clases se solape con el de las clases de las librerías que estemos utilizando. Para evitar esto es recomendable que utilicemos un mismo prefijo para todas nuestras clases. Las clases de Cocoa Touch tienen como prefijo dos letras mayúsculas que identifiquen cada librería y actúan como espacio de nombres (NS, CG, UI, etc), aunque no tenemos que utilizar por fuerza dos letras ni tampoco es obligatorio que sean mayúsculas. En los ejemplos estamos utilizando UA, ya que son las clases para una aplicación de la Universidad de Alicante. Esta nomenclatura también se aplica a otros símbolos globales, como estructuras, constantes, o funciones, pero nunca la utilizaremos para métodos ni variables de instancia. Esto es especialmente importante cuando estemos desarrollando librerías o frameworks que vayan a ser reutilizados.

Los métodos se escriben en *lowerCamelCase*. La composición del nombre resulta más compleja que en otros lenguajes, ya que se debe especificar tanto el nombre del método como el de sus parámetros. Si el método devuelve alguna propiedad, debería comenzar con el nombre de la propiedad. Si realiza alguna acción pondremos el verbo de la acción y su objeto directo si lo hubiese. A continuación deberemos poner los nombres de los parámetros. Para cada uno de ellos podemos anteponer una preposición (*From*, *With*, *For*, *To*, *By*, etc), y el nombre del parámetro, que podría ir precedido también de algún verbo. También podemos utilizar la conjunción *and* para separar los parámetros en el caso de que correspondan a una acción diferente.

Nota

En Objective-C es posible crear una clase que no herede de ninguna otra, ni siquiera de *NSObject*. Este tipo de clases se comportarán simplemente como estructuras de datos. También existe una segunda clase raíz diferente de *NSObject*, que es *NSProxy*, pero su uso es menos común (se utiliza para el acceso a objetos distribuidos). Normalmente siempre crearemos clases que en última instancia hereden de *NSObject*.

3.4.2. Implementación de la clase

La implementación se realiza en el fichero *.m*.

```
#import "UAAsignatura.h"

@implementation UAAsignatura
// Implementación de los métodos
@end
```

Vemos que siempre tenemos un `import` al fichero en el que se ha definido la interfaz. Deberemos añadir cualquier `import` adicional que necesitemos para nuestro código. Podemos implementar aquí los métodos definidos anteriormente:

```
#import "UAAsignatura.h"

const CGFloat UAHorasPorCredito = 10;
const CGFloat UADescuentoBecario = 0.5;

@implementation UAAsignatura

+ (CGFloat) creditosParaHoras:(CGFloat)horas
{
    return horas / UAHorasPorCredito;
}

- (CGFloat) creditos
{
    return [UAAsignatura creditosParaHoras: _horas];
}

- (CGFloat) tasaConPrecioPorCredito:(CGFloat)precioCredito
                                esBecario:(BOOL)becario
{
    CGFloat precio = [self creditos] * precioCredito;
    if(becario) {
        precio = precio * UADescuentoBecario;
    }
    return precio;
}

@end
```

Nota

Vemos que no es necesario añadir ningún `import` de las librerías básicas de Cocoa Touch porque ya están incluidos en el fichero `Prefix.pch` que contiene el prefijo común precompilado para todos nuestros fuentes.

Los métodos pueden recibir un número variable de parámetros. Para ello su firma deberá tener la siguiente forma:

```
+ (void)escribe:(NSInteger)n, ...;
```

Donde el valor `n` indica el número de parámetros recibidos, que pueden ser de cualquier tipo. En la implementación accederemos a la lista de los parámetros mediante el tipo `va_list` y una serie de macros asociadas (`va_start`, `va_arg`, y `va_end`):

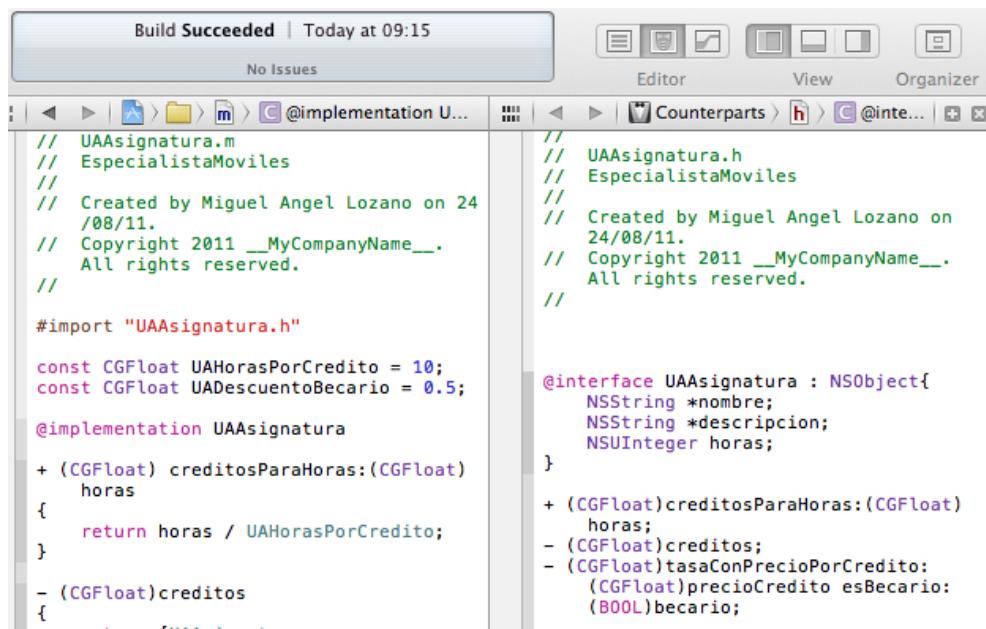
```
+ (void)escribe:(NSInteger *)numargs, ... {
    va_list parametros;
    va_start(parametros, numargs);
```

```

        for(int i=0; i<numargs; i++) {
            NSString *cad = va_arg(parametros,NSString *);
            NSLog(cad);
        }
        va_end(parametros);
    }
}

```

Hemos visto que para cada clase tenemos dos ficheros: el fichero de cabecera y el de implementación. Muchas veces necesitamos alternar entre uno y otro para añadir código o consultar lo que habíamos escrito en el fichero complementario. Para facilitar esta tarea, podemos aprovechar la vista del asistente en Xcode.



Vista del asistente

Podemos seleccionar el modo asistente mediante los botones que hay en la esquina superior derecha de la interfaz, concretamente el botón central del grupo *Editor*. Por defecto la vista asistente nos abrirá el fichero complementario al fichero seleccionado en la principal, pero esto puede ser modificado pulsando sobre el primer elemento de la barra de navegación del asistente. Por defecto vemos que está seleccionado *Counterparts*, es decir, el fichero complementario al seleccionado. Cambiando este elemento se puede hacer que se abran las subclases, superclases o cualquier otro elemento de forma manual.

Atajo

Podemos abrir directamente un fichero en el asistente si hacemos *Option(alt)-Click* sobre él en el panel del navegador.

Podemos añadir nuevas vistas de asistente, o eliminarlas, mediante los botones (+) y (X) que encontramos en su esquina superior derecha. También podemos cambiar la disposición del asistente mediante el menú *View > Assistant Editor*.

3.4.3. Creación e inicialización

Para instanciar una clase deberemos pasarle el mensaje `alloc` a la clase de la cual queramos crear la instancia. Por ejemplo:

```
id instancia = [NSString alloc];
```

Con esto crearemos una instancia de la clase `NSString`, reservando la memoria necesaria para alojarla, pero antes de poder utilizarla deberemos inicializarla con alguno de sus métodos inicializadores. Los inicializadores comienzan todos por `init`, y normalmente encontraremos definidos varios con distintos parámetros.

```
NSString *cadVacia = [[NSString alloc] init];
NSString *cadFormato = [[NSString alloc] initWithFormat: @"Numero %d", 5];
```

Como podemos ver, podemos anidar el paso de mensajes siempre que el resultado obtenido de pasar un mensaje sea un objeto al que podemos pasarle otro. Normalmente siempre encontraremos anidadas las llamadas a `alloc` e `init`, ya que son los dos pasos que siempre se deben dar para construir el objeto. Destacamos que `alloc` es un método de clase, que normalmente no será necesario redefinir, y que todas las clases heredarán de `NSObject` (en Objective-C las clases también son objetos y los métodos de clase se heredan), mientras que `init` es un método de instancia (se pasa el mensaje a la instancia creada con `alloc`), que nosotros normalmente definiremos en nuestras propias clases (de `NSObject` sólo se hereda un método `init` sin parámetros).

Sin embargo, en las clases normalmente encontramos una serie de métodos alternativos para instanciarlas y construirlas. Son los llamados métodos factoría, y en este caso todos ellos son métodos de clase. Suele haber un método factoría equivalente a cada uno de los métodos `init` definidos, y nos van a permitir instanciar e inicializar la clase directamente pasando un único mensaje. En lugar de `init`, comienzan con el nombre del objeto que están construyendo:

```
NSString *cadVacia = [NSString string];
NSString *cadFormato = [NSString stringWithFormat: @"Numero %d", 5];
```

Suelen crearse para facilitar la tarea al desarrollador. Estas dos formas de instanciar clases tienen una diferencia importante en cuanto a la gestión de la memoria, que veremos más adelante.

3.4.3.1. Implementación de inicializadores

Vamos a ver ahora cómo implementar un método de inicialización. Estos serán métodos devolverán una referencia de tipo `id` por convención, en lugar de devolver un puntero del tipo concreto del que se trate. Por ejemplo podemos declarar:

```
- (id)initWithNombre:(NSString*)nombre
               descripcion:(NSString*)descripcion
                  horas:(NSUInteger)horas;
```

Nota

En algunas ocasiones veremos los inicializadores declarados sin especificar ningún tipo de retorno (- `init`;). Esto es correcto, ya que en los métodos de Objective-C cuando no se declara tipo de retorno, por defecto se asume que es `id`.

Dentro de estos métodos lo primero que deberemos hacer es llamar al inicializador de la superclase, para que construya la parte del objeto relativa a ella. Si estamos heredando de `NSObject`, utilizaremos `[super init]` para inicializar esta parte, ya que es el único inicializador que define esa clase. Si heredamos de otra clase, podremos optar por otros inicializadores.

Una vez obtenido el objeto ya inicializado por la superclase, comprobaremos si la inicialización se ha podido hacer correctamente (mirando si el objeto devuelto es distinto de `nil`), y en ese caso realizaremos la inicialización pertinente. Finalmente devolveremos la referencia a nuestro objeto inicializado. Esta es la estructura que deberemos utilizar para los inicializadores de Objective-C:

```
- (id)initWithNombre:(NSString*)nombre
                 descripcion:(NSString*)descripcion
                     horas:(NSUInteger)horas
{
    self = [super init];
    if(self != nil) {
        _nombre = nombre;
        _descripcion = descripcion;
        _horas = horas;
    }
    return self;
}
```

Aquí vemos por primera vez el uso de las referencias `self` y `super`. La primera de ellas es una referencia a la instancia en la que estamos (equivalente a `this` en Java), mientras que la segunda es una referencia a la superclase, que nos permite llamar sobre ella a métodos que hayan podido ser sobrescritos.

Es importante reasignar la referencia `self` al llamar al constructor de la superclase, ya que en ocasiones es posible que el inicializador devuelva una instancia distinta a la instancia sobre la que se ejecutó. Siempre deberemos utilizar la referencia que nos devuelva el inicializador, no la que devuelva `alloc`, ya que podrían ser instancias distintas, aunque habitualmente no será así.

Nota

Tras la llamada a `alloc`, todas las variables de instancia de la clase se habrán inicializado a 0. Si este es el valor que queremos que tengan, no hará falta modificarlo en el inicializador.

3.4.3.2. Inicializador designado

Normalmente en una clase tendremos varios inicializadores, con distintos número de

parámetros. Por ejemplo en el caso anterior podríamos tener:

```
- (id)init;
- (id)initWithNombre:(NSString*)nombre;
- (id)initWithNombre:(NSString*)nombre
    descripcion:(NSString*)descripcion
    horas:(NSUInteger)horas;
```

La forma de inicializar el objeto será parecida, y por lo tanto podremos encontrar código repetido en estos tres inicializadores. Para evitar que ocurra esto, lo que deberemos hacer es establecer uno de nuestros inicializadores como inicializador designado. El resto de inicializadores siempre deberán recurrir a él para inicializar el objeto en lugar de inicializarlo por si mismos. Para que esto pueda ser así, este inicializador designado debería ser aquel que resulte más genérico, y que reciba un mayor número de parámetros con los que podamos inicializar explícitamente cualquier campo del objeto. Por ejemplo, en el caso anterior nuestro inicializador designado debería ser el que toma los parámetros `nombre`, `descripcion` y `horas`.

Los otros constructores podrán implementarse de la siguiente forma:

```
- (id)init
{
    return [self initWithNombre: @"Sin nombre"];
}

- (id)initWithNombre:(NSString *)nombre
{
    return [self initWithNombre:nombre
        descripcion:@"Sin descripcion"
        horas:-1];
}
```

Como vemos, no es necesario que todos ellos llamen directamente al inicializador designado, sino que pueden llamar a uno más genérico que ellos que a su vez llamará a otro más general hasta llegar al designado. De esta forma conseguimos un código más mantenible, ya que si queremos cambiar la forma de inicializar el objeto (por ejemplo si cambiamos el nombre de los campos), bastará con modificar el inicializador designado.

Deberemos también sobrescribir siempre el inicializador designado de la superclase (en nuestro ejemplo `init`), ya que si no lo hacemos se podría inicializar el objeto con dicho inicializador y nos estaríamos saltando el inicializador designado de nuestra clase, por lo que nuestras variables de instancia no estarían siendo inicializadas.

La única forma de conocer cuál es el inicializador designado de las clases de Cocoa Touch es recurrir a la documentación. En ocasiones el inicializador designado de una clase vendrá heredado por la superclase (en caso de que la subclase no defina constructores).

3.4.4. Gestión de la memoria

En Objective-C existen un sistema de recolección de basura como en el caso de Java, pero

lamentablemente no está disponible para la plataforma iOS, así que deberemos hacer la gestión de memoria de forma manual. No obstante, si seguimos una serie de reglas veremos que esto no resulta demasiado problemático. Además, contamos con los analizadores estático y dinámico que nos permitirán localizar posibles fugas de memoria (objetos instanciados que no llegan a ser liberados).

La gestión manual de la memoria se hace mediante la cuenta del número de referencias. Cuando llamamos a `alloc`, la cuenta de referencias se pone automáticamente a 1. Podemos incrementar número de referencias llamando `retain` sobre el objeto. Podemos decrementar el número de referencias llamando a `release`. Cuando el número de referencias llegue a 0, el objeto será liberado automáticamente de memoria.

Atención

Lo más importante es que el número de llamadas a `alloc` y `retain` sea el mismo que a `release`. Si faltase alguna llamada a `release` tendríamos una fuga de memoria, ya que habría objetos que no se eliminan nunca de memoria. Si por el contrario se llamase más veces de las necesarias a `release`, podríamos obtener un error en tiempo de ejecución por intentar acceder a una zona de memoria que ya no está ocupada por nuestro objeto.

La cuestión es: ¿cuándo debemos retener y liberar las referencias a objetos? La regla de oro es que **quien retiene debe liberar**. Es decir, en una de nuestras clases nunca deberemos retener un objeto y confiar en que otra lo liberará, y tampoco debemos liberar un objeto que no hemos retenido nosotros.

Cuando un objeto vaya a ser liberado de memoria (porque su número de referencias ha llegado a 0), se llamará a su método `dealloc` (destructor). En dicho método deberemos liberar todas las referencias que tenga retenidas el objeto, para evitar fugas de memoria. Nunca deberemos llamar a este método manualmente, será llamado por el sistema cuando el número de referencias sea 0.

En la inicialización de nuestro objeto habitualmente deberemos retener los objetos referenciados por nuestras variables de instancia, para evitar que dichos objetos puedan ser liberados de la memoria durante la vida de nuestro objeto.

```
- (id)initWithNombre:(NSString*)nombre
              descripcion:(NSString*)descripcion
                  horas:(NSUInteger)horas
{
    self = [super init];
    if(self != nil) {
        _nombre = [nombre retain];
        _descripcion = [descripcion retain];
        _horas = horas;
    }
    return self;
}
```

Todas aquellas referencias que hayan sido retenidas (conocidas como referencias fuertes), deberán ser liberadas antes de que nuestro objeto deje de existir. Para eso utilizaremos `dealloc`:

```
- (void)dealloc
{
    [_nombre release];
    [_descripcion release];
    [super dealloc];
}
```

Siempre deberemos llamar a `[super dealloc]` tras haber liberado las referencias retenidas de nuestras variables de instancia, para así liberar todo lo que haya podido retener la superclase.

3.4.4.1. Autorelease

Hemos visto que la regla que debemos seguir es que siempre deberá liberar las referencias quien las haya retenido. Pero a veces esto no es tan sencillo. Por ejemplo imaginemos un método que debe devolvernos una cadena de texto (que se genera dentro de ese mismo método):

```
- (NSString*) nombreAMostrar
{
    return [[NSString alloc] initWithFormat:@"%@", (_nombre, _horas)];
}
```

El método crea una nueva cadena, que tendrá el contador de referencias inicializado a 1 (tras llamar a `alloc`). Desde este método se devuelve la cadena y nunca más sabrá de ella, por lo que no podrá liberarla, lo cual hemos dicho que es responsabilidad suya. Tampoco podemos llamar a `release` antes de finalizar el método, ya que si hacemos eso quien reciba el resultado recibirá ya ese espacio liberado, y podría tener un error de acceso a memoria.

La solución es utilizar `autorelease`. Este método introduce un `release` pendiente para el objeto en un *pool* (conocido como *autorelease pool*). Estos `releases` pendientes no se ejecutarán hasta que haya terminado la pila de llamadas a métodos, por lo que podemos tener la seguridad de que al devolver el control nuestro método todavía no se habrá liberado el objeto. Eso sí, quien reciba el resultado, si quiere conservarlo, tendrá que retenerlo, y ya será responsabilidad suya liberarlo más adelante, ya que de no hacer esto, el objeto podría liberarse en cualquier momento debido al `autorelease` pendiente. La forma correcta de implementar el método anterior sería:

```
- (NSString*) nombreAMostrar
{
    return [[[NSString alloc] initWithFormat:@"%@", (_nombre, _horas)] autorelease];
}
```

Con esto cumplimos la norma de que quien lo retiene (`alloc`, `retain`), debe liberarlo (`release`, `autorelease`).

Cambios en iOS 5

En iOS 5 se introduce una característica llamada *Automatic Reference Counting* (ARC), que nos

libera de tener que realizar la gestión de la memoria. Si activamos esta característica, no deberemos hacer ninguna llamada a `retain`, `release`, o `autorelease`, sino que el compilador se encargará de detectar los lugares en los que es necesario realizar estas acciones y lo hará por nosotros. Por el momento hemos dejado esta opción deshabilitada, por lo que deberemos gestionar la memoria de forma manual. Más adelante veremos con más detalle el funcionamiento de ARC.

3.4.4.2. Métodos factoría

Antes hemos visto que a parte de los inicializadores, encontramos normalmente una serie de métodos factoría equivalentes. Por ejemplo a continuación mostramos un par inicializador-factoría que se puede encontrar en la clase `NSString`:

```
- (id)initWithFormat:(NSString *)format ...;
- (id)stringWithFormat:(NSString *)format ...;
```

El método factoría nos permite instanciar e inicializar el objeto con una sola operación, es decir, él se encargará de llamar a `alloc` y al inicializador correspondiente, lo cual facilita la tarea del desarrollador. Pero si seguimos la regla anterior, nos damos cuenta de que si es ese método el que está llamando a `alloc` (y por lo tanto reteniendo el objeto), deberá ser él también el que lo libere. Esto se hará gracias a `autorelease`. La implementación típica de estos métodos factoría es:

```
+ (id)asignaturaWithNombre:(NSString*)nombre
                      descripcion:(NSString*)descripcion
                        horas:(NSUInteger)horas
{
    return [[[UAAsignatura alloc] initWithNombre:nombre
                                         descripcion:descripcion
                                           horas:horas] autorelease];
}
```

Podemos observar que estos métodos son métodos de clase (nos sirven para crear una instancia de la clase), y dentro de ellos llamamos a `alloc`, al método `init` correspondiente, y a `autorelease` para liberar lo que hemos retenido, pero dando tiempo a que quien nos llame pueda recoger el resultado y retenerlo si fuera necesario.

3.4.4.3. Autorelease pool

Las llamadas a `autorelease` funcionan gracias a la existencia de lo que se conoce como *autorelease pool*, que va acumulando estos `releases` pendientes y en un momento dado los ejecuta. Podemos ver el *autorelease pool* de nivel superior en el fichero `main.m`:

```
int main(int argc, char *argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Podemos ver que la aplicación `UIApplication` se ejecuta dentro de un *autorelease pool*, de forma que cuando termine se ejecutarán todos los *autorelease* pendientes. Normalmente no deberemos preocuparnos por eso, ya que la API de Cocoa habrá creado *autorelease pools* en los lugares apropiados (en los eventos que llaman a nuestro código), pero en alguna ocasión puede ser necesario crear nuestros propios *autorelease pools*. Por ejemplo, en el caso en que tengamos un bucle que itera un gran número de veces y que en cada iteración crea objetos con *autorelease*s pendientes. Para evitar quedarnos sin memoria, podemos introducir un *autorelease pool* propio dentro del bucle:

```
for(int i=0; i<n; i++) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSString *cadena = [NSString string];
    ...
    [pool release];
}
```

De esta forma estamos haciendo que en cada iteración del bucle se liberen todos los *autorelease*s pendientes que hubiese. Cuando llamemos a `autorelease`, siempre se utilizará el *pool* de ámbito más cercano que exista.

3.4.4.4. Patrón singleton

Para implementar el patrón *singleton* en Objective-C necesitaremos un método factoría que nos proporcione la instancia única del objeto, y una variable de tipo `static` que almacene dicha instancia. El método factoría tendrá el siguiente aspecto:

```
+ (UADatosCompartidos) sharedDatosCompartidos {
    static DatosCompartidos *datos = nil;
    if(nil == datos) {
        datos = [[DatosCompartidos alloc] init];
    }
    return datos;
```

Nota

Con esto no estamos impidiendo que alguien pueda crear nuevas instancias del objeto llamando directamente a `alloc`, en lugar de pasar por la factoría. Para evitar esto, podríamos sobrescribir el método `allocWithZone` para que sólo instancie el objeto una única vez, y también podríamos sobrescribir los métodos de gestión de memoria para inhabilitarlos (`retain`, `release` y `autorelease`).

3.5. Protocolos

Los protocolos son el equivalente a las interfaces en Java. Definen una serie de métodos, que los objetos que los adopten tendrán que implementar. Los protocolos se definen de la siguiente forma:

```
@protocol UACalificable
- (void)setNota: (CGFloat) nota;
- (CGFloat)nota;
@end
```

Para hacer que una clase adopte lo especificaremos entre <...> tras el nombre de la superclase en la declaración de nuestra clase:

```
@interface UAAsignatura : NSObject<UACalificable>
...
@end
```

Podemos utilizar el tipo referencia genérica `id` junto a un nombre de protocolo para referenciar cualquier clase que adopte el protocolo, independientemente de su tipo concreto, y que el compilador verifique que dicha clase al menos define los métodos declarados en el protocolo (en caso contrario obtendríamos un *warning*):

```
id<UACalificable> objetoCalificable;
```

Los protocolos en Objective-C nos permiten definir tanto métodos de implementación obligatoria (por defecto) como opcional:

```
@protocol MiProtocolo
- (void)metodoObligatorio;
@optional
- (void)metodoOpcional;
- (void)otroMetodoOpcional;
@required
- (void)otroMetodoObligatorio;
@end
```

Los nombres de los protocolos, al igual que las clases, se deberán escribir en notación `UpperCamelCase`, pero en este caso se recomienda utilizar gerundios (p.ej. `NSCopying`) o adjetivos.

3.6. Categorías y extensiones

Las categorías nos permiten añadir métodos adicionales a clases ya existentes, sin necesidad de heredar de ellas ni de modificar su código. Para declarar una categoría especificaremos la clase que extiende y entre paréntesis el nombre de la categoría. En la categoría podremos declarar una serie de métodos que se añadirán a los de la clase original, pero no podemos añadir variables de instancia:

```
@interface NSString ( CuentaCaracteres )
- (NSUInteger)cuentaCaracter:(char)caracter;
@end
```

La implementación se definirá de forma similar. El fichero en el que se almacenan las categorías suele tomar como nombre `NombreClase+NombreCategoria.h` (`.m`). Por ejemplo, en nuestro caso se llamaría `NSString+CuentaCaracteres`:

```
#import "NSString+CuentaCaracteres.h"
@implementation NSString ( CuentaCaracteres )
- (NSUInteger)cuentaCaracter:(char)caracter { ... }
```

```
@end
```

Las categorías resultarán de utilidad para incorporar métodos de utilidad adicionales que nuestra aplicación necesite a las clases que correspondan, en lugar de tener que crear clases de utilidad independientes (como el ejemplo en el que hemos extendido `NSString`). También serán útiles para poder repartir la implementación de clases complejas en diferentes ficheros.

Por otro lado tenemos lo que se conoce como extensiones. Una extensión se define como una categoría sin nombre, y nos fuerza a que los métodos que se definen en ella estén implementados en su clase correspondiente. Esto se usa frecuentemente para declarar métodos privados en nuestras clases. Anteriormente hemos comentado que en Objective-C todos los métodos son públicos, pero podemos tener algo similar a los métodos privados si implementamos métodos que no estén declarados en el fichero de cabecera público de nuestra clase. Estos métodos no serán visibles por otras clases, y si intentamos acceder a ellos el compilador nos dará un *warning*. Sin embargo, en tiempo de ejecución el acceso si que funcionará ya que el método realmente está implementado y como hemos comentado, realmente el acceso privado a métodos no existe en este lenguaje.

El problema que encontramos haciendo esto es que si nuestro método no está declarado en ningún sitio, será visible por los métodos implementados a continuación del nuestro, pero no los que aparecen anteriormente en el fichero de implementación. Para evitar esto deberíamos declarar estos métodos al comienzo de nuestro fichero de implementación, para que sea visible por todos los métodos. Para ello podemos declarar una extensión en el fichero de implementación:

```
@interface MiObjeto ()
- (void)metodoPrivado;
@end

@implementation MiObjeto

// Otros métodos
...
- (void)metodoPrivado {
    ...
}
@end
```

Atención

No deberemos utilizar el prefijo `_` para los métodos privados. Esta nomenclatura se la reserva Apple para los métodos privados de las clases de Cocoa Touch. Si la utilizásemos, podríamos estar sobrescribiendo por error métodos de Cocoa que no deberíamos tocar.

3.7. Algunas clases básicas de Cocoa Touch

Vamos a empezar viendo algunos ejemplos de clases básicas de la API Cocoa Touch que necesitaremos para implementar nuestras aplicaciones. En estas primeras sesiones comenzaremos con el *framework Foundation*, donde tenemos la librería de clases de propósito general, y que normalmente encontraremos con el prefijo `NS`.

3.7.1. Objetos

`NSObject` es la clase de la que normalmente heredarán todas las demás clases, por lo que sus métodos estarán disponibles en casi en todas las clases que utilicemos. Vamos a ver los principales métodos de esta clase.

3.7.1.1. Inicialización e instantiación

Ya conocemos algunos métodos de este grupo (`alloc` e `init`), pero podemos encontrar alguno más:

- + `initialize`: Es un método de clase, que se llama cuando la clase se carga por primera vez, antes de que cualquier instancia haya sido cargada. Nos puede servir para inicializar variables estáticas.
- + `new`: Este método lo único que hace es llamar a `alloc` y posteriormente a `init`, para realizar las dos operaciones con un único mensaje. No se recomienda su uso.
- + `allocWithZone`: (`NSZone*`): Reserva memoria para el objeto en la zona especificada. Si pasamos `nil` como parámetro lo aloja en la zona por defecto. El método `alloc` visto anteriormente realmente llama a `allocWithZone: nil`, por lo que si queremos cambiar la forma en la que se instancia el objeto, con sobrescribir `allocWithZone` sería suficiente (esto se puede utilizar por ejemplo al implementar el patrón *singleton*, para evitar que el objeto se instancie más de una vez). Utilizar zonas adicionales puede permitirnos optimizar los accesos memoria (para poner juntos en memoria objetos que vayan a utilizarse de forma conjunta), aunque normalmente lo más eficiente será utilizar la zona creada por defecto.
- - `copy` / - `mutableCopy`: Son métodos de instancia que se encargan de crear una nueva instancia del objeto copiando el estado de la instancia actual. Estos métodos pondrán a uno el contador de referencias de la nueva instancia (pertenece al grupo de métodos que incrementan este contador, junto a `alloc` y `retain`). No todos los objetos son copiables, a continuación veremos más detalles sobre la copia de objetos.

3.7.1.2. Copia de objetos

La clase `NSObject` incorpora el método `copy` que se encarga de crear una copia de nuestro objeto. Sin embargo, para que dicho método funcione necesita que esté implementado el método `copyWithZone` que no está en `NSObject`, sino en el protocolo `NSCopying`. Por lo tanto, sólo los objetos que adopten dicho protocolo serán copiables.

Gran parte de los objetos de la API de Cocoa Touch implementan `NSCopying`, lo cual quiere decir que son copiables. Si queremos implementar este protocolo en nuestros

propios objetos, la forma de implementarlo dependerá de si nuestra superclase ya implementaba dicho protocolo o no:

- Si la superclase no implementa el protocolo, deberemos implementar el método `copyWithZone` de forma que dentro de él se llame a `allocWithZone` (o `alloc`) para crear una nueva instancia de nuestro objeto y al inicializador correspondiente para inicializarlo con los datos de nuestro objeto actual. Además deberemos copiar los valores de las variables de instancia que no hayan sido asignadas con el inicializador.

```
- (id)copyWithZone:(NSZone *)zone
{
    return [[UAAsignatura allocWithZone:zone]
            initWithNombre:_nombre
            descripcion:_descripcion
            horas:_horas];
}
```

- Si nuestra superclase ya era copiable, entonces deberemos llamar al método `copyWithZone` de la superclase y una vez hecho esto copiar los valores de las variables de instancia definidas en nuestra clase, que no hayan sido copiadas por la superclase.

```
- (id)copyWithZone:(NSZone *)zone
{
    id copia = [super copyWithZone: zone];
    // Copiar propiedades de nuestra clase
    [copia setNombre: _nombre];
    ...
    return copia;
}
```

Otra cosa que debemos tener en cuenta al implementar las copias, es si los objetos son **mutables** o **inmutables**. En la API de Cocoa encontramos varios objetos que se encuentran disponibles en ambas versiones, como por ejemplo las cadenas (`NSString` y `NSMutableString`). Los objetos inmutables son objetos de los cuales no podemos cambiar su estado interno, y que una vez instanciados e inicializados, sus variables de instancia no cambiarán de valor. Por otro lado, los mutables son aquellos cuyo estado si puede ser modificado.

En caso de que nuestro objeto sea **inmutable**, podemos optimizar la forma de hacer las copias. Si el estado del objeto no va a cambiar, no es necesario crear una nueva instancia en memoria. Obtendremos el mismo resultado si en el método `copy` nos limitamos a retener el objeto con `retain` (esto es necesario ya que siempre se espera que `copy` retenga el objeto).

Los objetos que existan en ambas modalidades (mutable e inmutable) pueden adoptar también el protocolo `NSTMutableCopy`, que nos obligará a implementar el método `mutableCopyWithZone`, permitiéndonos utilizar `mutableCopy`, para así poder hacer la copia de las dos formas vistas anteriormente. En estos objetos `copy` realizará la copia inmutable simplemente reteniendo nuestro objeto, mientras que `mutableCopy` creará una copia mutable como una nueva instancia.

Un objeto que existe en ambas modalidades es por ejemplo las cadenas: `NSString` y `NSMutableString`. Si de un objeto `NSString` hacemos un `copy`, simplemente se estará reteniendo el mismo objeto. Si hacemos `mutableCopy` estaremos creando una copia del objeto, que además será de tipo `NSMutableString`.

```
NSString *cadena = @"Mi cadena";  
  
// copiaInmutable==cadena  
NSString *copiaInmutable = [cadena copy];  
  
// copiaMutable!=cadena  
NSMutableString *copiaMutable = [cadena mutableCopy];
```

Si nuestro objeto original fuese una cadena mutable, cualquiera de los dos métodos de copia creará una nueva instancia. La copia inmutable creará un instancia de tipo `NSString` que no podrá ser alterada, mientras que la copia mutable creará una nueva instancia de tipo `NSMutableString`, que se podrá alterar sin causar efectos laterales con nuestro objeto original.

```
NSMutableString *cadenaMutable = [NSMutableString stringWithCapacity: 32];  
  
// copiaInmutable!=cadena  
NSString *copiaInmutable = [cadena copy];  
  
// copiaMutable!=cadena  
NSMutableString *copiaMutable = [cadena mutableCopy];
```

3.7.1.3. Información de la instancia

Al igual que ocurría en Java, `NSObject` implementa una serie de métodos que nos darán información sobre los objetos, y que nosotros podemos sobrescribir en nuestra clases para personalizar dicha información. Estos métodos son:

- `isEqual`: Comprueba si dos instancias de nuestra clase son iguales internamente, y nos devuelve un *booleano* (YES o NO).
- `description`: Nos da una descripción de nuestro objeto en forma de cadena de texto (`NSString`).
- `hash`: Genera un código *hash* a partir de nuestro objeto para indexarlo en tablas. Si hemos redefinido `isEqual`, deberíamos también redefinir `hash` para que dos objetos iguales generen siempre el mismo *hash*.

3.7.2. Cadenas

La clase `NSString` es la clase con la que representamos las cadenas en Objective-C, y hemos visto cómo utilizarla en varios de los ejemplos anteriores. Vamos ahora a ver algunos elementos básicos de esta clase.

3.7.2.1. Literales de tipo cadena

La forma más sencilla de inicializar una cadena es utilizar un literal de tipo `@"cadena"`,

que inicializa un objeto de tipo `NSString*`, y que no debemos confundir con las cadenas de C estándar que se definen como "cadena" (sin la @) y que corresponden al tipo `char*`.

Estos literales tienen la peculiaridad de que se crean de forma estática por el compilador, por lo que las operaciones de gestión de memoria no tienen efecto sobre ellos (nunca serán eliminados de la memoria). Las llamadas que realicemos sobre estos objetos a `retain`, `release` y `autorelease` serán ignoradas.

3.7.2.2. Cadenas C estándar y Objective-C

Como hemos comentado, las cadenas @"cadena" y "cadena" son tipos totalmente distintos, por lo que no podemos utilizarlas en las mismas situaciones. Las clases de Cocoa Touch siempre trabajarán con `NSString`, por lo que si tenemos cadenas C estándar tendremos que convertirlas previamente. Para ello, en la clase `NSString` contamos con métodos inicializadores que crear la cadena Objective-C a partir de una cadena C:

```
- (id)initWithCString:(const char *)nullTerminatedCString
                  encoding:(NSStringEncoding)encoding
- (id)initWithUTF8String:(const char *)bytes
```

El primero de ellos inicializa la cadena Objective-C a partir de una cadena C y de la codificación que se esté utilizando en dicha cadena. Dado que la codificación más común es UTF-8, tenemos un segundo método que la inicializa considerando directamente esta codificación. También encontramos métodos de factoría equivalentes:

```
- (id)stringWithCString:(const char *)nullTerminatedCString
                  encoding:(NSStringEncoding)encoding
- (id)stringWithUTF8String:(const char *)bytes
```

De la misma forma, puede ocurrir que tengamos una cadena en Objective-C y que necesitemos utilizarla en alguna función C estándar. En la clase `NSString` tenemos métodos para obtener la cadena como cadena C estándar:

```
NSString *cadena = @"Cadena";
const char *cadenaC = [cadena UTF8String];
const char *cadenaC = [cadena cStringUsingEncoding: NSASCIIStringEncoding];
```

3.7.2.3. Inicialización con formato

Podemos dar formato a las cadenas de forma muy parecida al `printf` de C estándar. Para ello contamos con el inicializador `initWithFormat`:

```
NSString *cadena = [[NSString alloc]
                     initWithFormat: @"Duracion: %d horas", horas];
```

A los códigos de formato que ya conocemos de `printf` en C, hay que añadir %@ que nos permite imprimir objetos Objective-C. Para imprimir un objeto utilizará su método `description` (o `descriptionWithLocale` si está implementado). Deberemos utilizar dicho código para imprimir cualquier objeto Objective-C, incluido `NSString`:

```
NSString *nombre = @"Pepe";
```

```
NSUInteger edad = 20;  
NSString *cadena =  
    [NSString stringWithFormat: @"Nombre: %@", (edad %d), nombre, edad];
```

Atención

Nunca se debe usar el código `%s` con una cadena Objective-C (`NSString`). Dicho código espera recibir un *array* de caracteres acabado en `NULL`, por lo que si pasamos un puntero a objeto obtendremos resultados inesperados. Para imprimir una cadena Objective-C siempre utilizaremos `%@`.

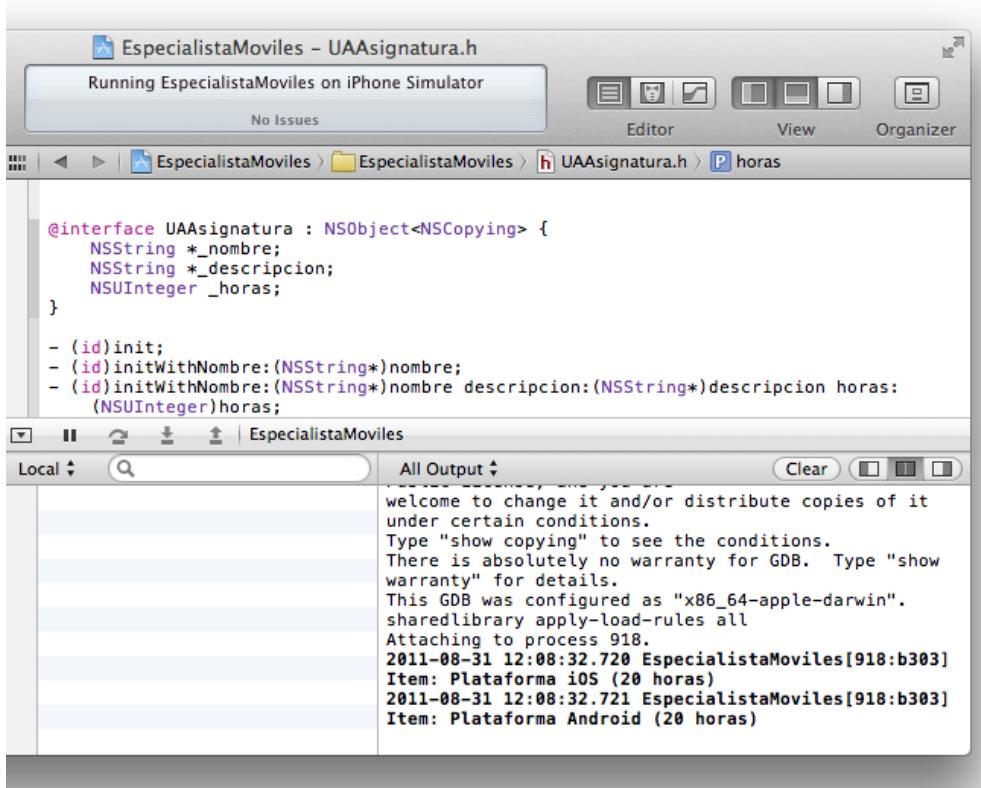
Los mismos atributos de formato se podrán utilizar en la función `NSLog` que nos permite escribir *logs* en la consola para depurar aplicaciones

```
NSLog(@"%@", i = %d, obj = %@", i, obj);
```

Cuidado

Los *logs* pueden resultarnos muy útiles para depurar la aplicación, pero debemos llevar cuidado de eliminarlos en la *release*, ya que reducen drásticamente el rendimiento de la aplicación, y tampoco resulta adecuado que el usuario final pueda visualizarlos. Podemos ayudarnos de las macros (`#define`, `#ifdef`) para poder activarlos o desactivarlos de forma sencilla según la configuración utilizada.

Los *logs* aparecerán en el panel de depuración ubicado en la parte inferior de la pantalla principal del entorno. Podemos abrirlo y cerrarlo mediante en botón correspondiente en la esquina superior izquierda de la pantalla. Normalmente cuando ejecutemos la aplicación y ésta escriba *logs*, dicho panel se mostrará automáticamente.



Panel de depuración

3.7.2.4. Localización de cadenas

Anteriormente hemos comentado que las cadenas se puede externalizar en un fichero que por defecto se llamará `Localizable.strings`, del que podremos tener varias versiones, una para cada localización soportada. Vamos a ver ahora cómo leer estas cadenas localizadas. Para leerlas contamos con la función `NSLocalizedString(clave, comentario)` que nos devuelve la cadena como `NSString`:

```
NSString *cadenaLocalizada = NSLocalizedString(@"Titulo", @"Mobile UA");
```

Este método nos devolverá el valor asociado a la clave proporcionada según lo especificado en el fichero `Localizable.strings` para el *locale* actual. Si no se encuentra la clave, nos devolverá lo que hayamos especificado como comentario.

Existen versiones alternativas del método que no utilizan el fichero por defecto, sino que toman como parámetro el fichero del que sacar las cadenas.

Las cadenas de `Localizable.strings` pueden también contener códigos de formato. En estos casos puede ser interesante numerar los parámetros, ya que puede que en diferentes idiomas el orden sea distinto:

```
// es.lproj
```

```
"CadenaFecha" = "Fecha: %1$2d / %2$2d / %3$4d";
// en.lproj
"CadenaFecha" = "Date: %2$2d / %1$2d / %3$4d";
```

Podemos utilizar `NSLocalizedString` para obtener la cadena con la plantilla del formato:

```
NSString *cadena = [NSString stringWithFormat:
    NSLocalizedString(@"CadenaFecha", "Date: %2$2d / %1$2d / %3$4d"),
    dia, mes, año];
```

3.7.2.5. Conversión de números

La conversión entre cadenas y los diferentes tipos numéricos en Objective-C también se realiza con la clase `NSString`. La representación de un número en forma de cadena se puede realizar con el método `stringWithFormat` visto anteriormente, permitiendo dar al número el formato que nos interese.

La conversión inversa se puede realizar mediante una serie de métodos de la clase `NSString`:

```
NSInteger entero = [cadenaInt integerValue];
BOOL booleano = [cadenaBool boolValue];
float flotante = [cadenaFloat floatValue];
```

3.7.2.6. Comparación de cadenas

Las cadenas son punteros a objetos, por lo que si queremos comparar si dos cadenas son iguales nunca deberemos utilizar el operador `==`, ya que esto sólo comprobará si los dos punteros apuntan a la misma dirección de memoria. Para comparar si dos cadenas contienen los mismos caracteres podemos utilizar el método `isEqual`, al igual que para comparar cualquier otro tipo de objeto Objective-C, pero si sabemos que los dos objetos son cadenas es más sencillo utilizar el método `isEqualToString`.

```
if([cadena isEqualToString: otraCadena]) { ... }
```

También podemos comparar dos cadenas según el orden alfabético, con `compare`. Nos devolverá un valor de la enumeración `NSComparisonResult` (`NSOrderedAscending`, `NSOrderedSame`, o `NSOrderedDescending`).

```
NSComparisonResult resultado = [cadena compare: otraCadena];
switch(resultado) {
    case NSOrderedAscending:
        ...
        break;
    case NSOrderedSame:
        ...
        break;
    case NSOrderedDescending:
        ...
        break;
}
```

Tenemos también el método `caseInsensitiveCompare` para que realice la comparación ignorando mayúsculas y minúsculas.

Otros métodos nos permiten comprobar si una cadena tiene un determinado prefijo o sufijo (`hasPrefix`, `hasSuffix`), u obtener la longitud de una cadena (`length`).

3.7.3. Fechas

Otro tipo de datos que normalmente necesitaremos tratar en nuestras aplicaciones son las fechas. En Objective-C las fechas se encapsulan en `NSDate`. Muchos de los métodos de dicha clase toman como parámetro datos del tipo `NSTimeInterval`, que equivale a `double`, y corresponde al tiempo en segundos (con una precisión de submilisegundos).

La forma más rápida de crear un objeto fecha es utilizar su inicializador (o factoría) sin parámetros, con lo que se creará un objeto representando la fecha actual.

```
NSDate *fechaActual = [NSDate date];
```

También podemos crear la fecha especificando el número de segundos desde la fecha de referencia (1 de enero de 1970 a las 0:00).

```
NSDate *fecha = [NSDate dateWithTimeIntervalSince1970: segundos];
```

De la misma forma que en el caso de las cadenas, podemos comparar fechas con el método `compare`, que también nos devolverá un valor de la enumeración `NSComparisonResult`.

3.7.3.1. Componentes de la fecha

El objeto `NSDate` representa una fecha simplemente mediante el número de segundos transcurridos desde la fecha de referencia. Sin embargo, muchas veces será necesario obtener los distintos componentes de la fecha de forma independiente (día, mes, año, hora, minutos, segundos, etc). Estos componentes se encapsulan como propiedades de la clase `NSDateComponents`.

Para poder obtener los componentes de una fecha, o crear una fecha a partir de sus componentes, necesitamos un objeto calendario `NSCalendar`:

```
NSDate *fecha = [NSDate date];
NSCalendar *calendario = [NSCalendar currentCalendar];
NSDateComponents *componentes = [currentCalendar
    components:(NSDayCalendarUnit | NSMonthCalendarUnit |
    NSYearCalendarUnit)
    fromDate:fecha];

NSInteger dia = [componentes day];
NSInteger mes = [componentes month];
NSInteger anyo = [componentes year];
```

Con el método de clase `currentCalendar` obtenemos una instancia del calendario correspondiente al usuario actual. Con el método `components:fromDate:` de dicho

calendario podemos extraer los componentes indicados de la fecha (objeto `NSDate`) que proporcionemos. Los componentes se especifican mediante una máscara que se puede crear a partir de los elementos de la enumeración `NSCalendarUnit`, y son devueltos mediante un objeto de tipo `NSDateComponents` que incluirá los componentes solicitados como campos.

También se puede hacer al contrario, crear un objeto `NSDateComponents` con los campos que queramos para la fecha, y a partir de él obtener un objeto `NSDate`:

```
NSDateComponents *componentes = [[NSDateComponents alloc] init];
[componentes setDay: dia];
[componentes setMonth: mes];
[componentes setYear: anyo];

NSDate *fecha = [calendario dateFromComponents: componentes];
[componentes release];
```

Con el calendario también podremos hacer operaciones con fechas a partir de sus componentes. Por ejemplo, podemos sumar valores a cada componentes con `dateByAddingComponents:toDate:`, o obtener la diferencia entre dos fechas componente a componente con `components:fromDate:toDate:options:`.

3.7.3.2. Formato de fechas

Podemos dar formato a las fecha con `NSDateFormatter`. La forma más sencilla es utilizar alguno de los estilos predefinidos en la enumeración `NSDateFormatterStyle` (`NSDateFormatterNoStyle`, `NSDateFormatterShortStyle`, `NSDateFormatterMediumStyle`, `NSDateFormatterLongStyle`, `NSDateFormatterFullStyle`):

```
NSDateFormatter formato = [[NSDateFormatter alloc] init];
[formato setTimeStyle: NSDateFormatterNoStyle];
[formato setDateStyle: NSDateFormatterFullStyle];

NSString *cadena = [formato stringFromDate: fecha];
[formato release];
```

Podemos también especificar un formato propio mediante un patrón con `setDateFormat:`

```
[formato setDateFormat: @"dd/MM/yyyy HH:mm"];
```

También podremos utilizar nuestro objeto de formato para *parsear* fechas con `dateFromString:`

```
NSDate *fecha = [formato dateFromString: @"20/06/2012 14:00"];
```

3.7.4. Errores y excepciones

En Objective-C podemos tratar los errores mediante excepciones de forma similar a Java. Para capturar una excepción podemos utilizar la siguiente estructura:

```

@try
    // Código
@catch(NSErrorException *ex) {
    // Código tratamiento excepción
}
@catch(id obj) {
    // Código tratamiento excepción
}
@finally {
    // Código de finalización
}

```

Una primera diferencia que encontramos con Java es que se puede lanzar cualquier objeto (por ejemplo, en el segundo `catch` vamos que captura `id`), aunque se recomienda utilizar siempre `NSErrorException` (o una subclase de ésta). Otra diferencia es que en Objective-C no suele ser habitual heredar de `NSErrorException` para crear nuestros propios tipos de excepciones. Cuando se produzca una excepción en el código del bloque `try` saltará al primer `catch` cuyo tipo sea compatible con el del objeto lanzado. El bloque `finally` siempre se ejecutará, tanto si ha lanzado la excepción como si no, por lo que será el lugar idóneo para introducir el código de finalización (por ejemplo, liberar referencias a objetos).

Podremos lanzar cualquier objeto con `@throw`:

```

@throw [[[NSError alloc] initWithName: @"Error"
                                reason: @"Descripción del error"
                               userInfo: nil] autorelease];

```

Aunque tenemos disponible este mecanismo para tratar los errores, en Objective-C suele ser más común pasar un parámetro de tipo `NSError` a los métodos que puedan producir algún error. En caso de que se produzca, en dicho objeto tendremos la descripción del error producido:

```

NSError *error;
NSString *contenido = [NSString
    stringWithContentsOfFile: @"texto.txt"
    encoding: NSASCIIStringEncoding
    error: &error];

```

Este tipo de métodos reciben como parámetro la dirección del puntero, es decir, (`NSError **`), por lo que no es necesario que inicialicemos el objeto `NSError` nosotros. Si no nos interesa controlar los errores producidos, podemos pasar el valor `nil` en el parámetro `error`. Los errores llevan principalmente un código (`code`) y un dominio (`domain`). Los código se definen como constantes en Cocoa Touch (podemos consultar la documentación de `NSError` para consultarlos). También incorpora mensajes que podríamos utilizar para mostrar el motivo del error y sus posibles soluciones:

```

NSString *motivo = [error localizedFailureReason];

```

En Objective-C no hay ninguna forma de crear excepciones equivalentes a las excepciones de tipo *checked* en Java (es decir, que los métodos estén obligados a capturarlas o a declarar que pueden lanzarlas). Por este motivo, aquellos métodos en los que en Java utilizaríamos excepciones *checked* en Objective-C no será recomendable

utilizar excepciones, sino incorporar un parámetro `NSError` para así dejar claro en la interfaz que la operación podría fallar. Sin embargo, las excepciones si que serán útiles si queremos tratar posibles fallos inesperados del *runtime* (las que serían equivalentes a las excepciones *unchecked* en Java).

Acceso a la documentación

Como hemos comentado, mientras escribimos código podemos ver en el panel de utilidades ayuda rápida sobre el elemento sobre el que se encuentre el cursor en el editor. Tenemos también otros atajos para acceder a la ayuda. Si hacemos *option(alt)-click* sobre un elemento del código abriremos un cuadro con su documentación, a partir del cual podremos acceder a su documentación completa en *Organizer*. Por otro lado, si hacemos *cmd-click* sobre un elemento del código, nos llevará al lugar en el que ese elemento fue definido. Esto puede ser bastante útil para acceder de forma rápida a la declaración de clases y de tipos.

4. Ejercicios de Objective-C

4.1. Manejo de cadenas

Vamos a cambiar las cadenas de la aplicación que comenzamos a desarrollar en la sesión anterior. Se pide:

- a) En la clase `UAMasterViewController` localiza el método `initWithNibName:bundle:`. Hay una línea en la que se asigna el título mediante la instrucción `self.title =` Modifica el título para que ahora sea "Filmoteca".
- b) Abre la documentación de la clase `NSString`. Busca un método que permita convertir la cadena a mayúsculas, y aplícalo al título.
- c) Localiza el título de la aplicación utilizando la cadena con identificador "`AppName`" definida en la sesión anterior.
- d) Vamos a hacer que en la lista aparezcan varios ítems. Para ello localiza el método `tableView:numberOfRowsInSection:`, y haz que devuelva el valor 5. Comprueba que ahora aparecen cinco ítems en la lista.
- e) Ahora cambiaremos el texto de cada ítem, para que indique la posición de la lista en la que está. Por ejemplo, el primer ítem tendrá el texto Posición 0, el segundo Posición 1, etc. Para ello localiza el método `tableView:cellForRowAtIndexPath:`, y dentro de él la línea que comienza por `cell.textLabel.text =` Aquí es donde se asigna el texto de cada ítem. La posición del ítem que se está asignando se puede obtener llamando al método `row` del objeto `NSIndexPath`, y es un valor de tipo entero (consulta la documentación de la clase `NSIndexPath` en Organizer).

4.2. Creación de objetos

Vamos a crear una clase para representar las películas. La clase tendrá como nombre `UAPelicula`, y contendrá los siguientes datos:

- Título
- Director
- Calificación de edad (puede ser TP, NR7, NR13 o NR18)
- Puntuación (número real de 0 a 10)

- a) Crea la clase, e introduce las variables de instancia necesarias para representar los datos anteriores, utilizando el tipo de datos que consideres más adecuado.
- b) Implementa un inicializador sin parámetros, otro que lleve sólo el título, y otro que incluya todos los datos. ¿Cuál será el inicializador designado? Implémentalos teniendo esto en cuenta.

- c) Crea un método factoría para cada inicializador.
- d) Sobrescribe el método `description` para que imprima el título de la película, seguido del nombre del director entre paréntesis. Por ejemplo: `El Resplandor (Stanley Kubrick)`.
- e) Modifica la clase `UAMasterViewController` para que en su inicializador se instancie un objeto de tipo película, y en su `dealloc` se libere su memoria. Como texto de las celdas, mostraremos la descripción del objeto `UAPelicula` creado.

4.3. Manejo de fechas (*)

Vamos a añadir a la clase `UAPelicula` la fecha de estreno de la película. Se pide:

- a) Añadir una nueva variable de instancia para la fecha de estreno de la película. Modificaremos los inicializadores para incluir este dato adicional en el inicializador designado.
- b) Modificamos el método `description` para que la película ahora muestre junto al nombre del director, el año de su estreno. Por ejemplo, `El Resplandor (1980, Stanley Kubrick)`. En el campo fecha realmente tendremos almacenada la fecha exacta (día, mes y año).
- c) Añadir un método que nos devuelva la antigüedad de la película en años. Por ejemplo, si la película se estrenó en 1980 y estamos en 2011, nos devolverá 31 años. Escribir un `log` con la antigüedad tras instanciar la clase en el inicializador de `UAMasterViewController`.

4.4. Gestión de errores (*)

Vamos a ver como tratar errores al llamar a métodos que pueden fallar por factores externos. Utilizaremos el método `stringWithContentOfFile:encoding:error:` de la clase `NSString` para este ejercicio. Deberemos consultar la documentación de este método en Organizer. Se pide:

- a) Crear dos ficheros de texto: `texto1.txt` y `texto2.txt` que contengan el texto `Primer fichero` y `Segundo fichero` respectivamente. El primero de ellos lo empaquetaremos en el raíz del *bundle*, y el segundo en una carpeta `/datos`.

Ayuda

Para obtener la ruta de un fichero en el sistema de ficheros del dispositivos, puedes utilizar el método `[[NSBundle mainBundle] pathForResource:@"nombreFichero" ofType:@"extension"]`.

- b) En el inicializador de `UAMasterViewController` crearemos una cadena a partir de

estos ficheros, y escribiremos un *log* con el contenido leído.

c) Intenta leer el segundo fichero con la ruta `/texto2.txt`. Deberá dar error, ya que este fichero se empaqueta en `/datos/texto2.txt`. Utiliza el objeto `NSError` para detectar el error y escribe un *log* con el motivo del error.

5. Propiedades, colecciones y gestión de la memoria

5.1. Propiedades de los objetos

Como hemos visto, las variables de instancia de los objetos normalmente son protegidas y accederemos a ellas a través de métodos accesores (*getters* y *setters*). La forma de escribir estos métodos será siempre la misma, por lo que suele resultar una tarea bastante repetitiva y anodina, que puede ser realizada perfectamente de forma automática. Por ejemplo, en Java normalmente los IDEs nos permiten generar automáticamente estos métodos.

En Objective-C contamos con las denominadas propiedades, que realmente son una forma de acceso que nos da el lenguaje a las variables de instancia sin que tengamos que implementar manualmente los métodos accesores.

La propiedades se definen dentro de la declaración de la interfaz mediante la etiqueta `@property`. La etiqueta puede tomar una serie de atributos, como por ejemplo `nonatomic`, que indica que los métodos accesores no deben ser atómicos, es decir, que no se debe sincronizar el acceso a ellos desde diferentes hilos. Normalmente todas las propiedades serán de este tipo, ya que no es frecuente que podamos tener problemas de concurrencia en las aplicaciones que desarrollaremos habitualmente, y sincronizar el acceso tiene un elevado coste en rendimiento. Tras `@property` y sus atributos declararemos el tipo y el nombre de la propiedad.

```
@interface UAAsignatura : NSObject{
    NSString *_nombre;
    NSString *_descripcion;
    NSUInteger _horas;
}

- (id)init;
- (id)initWithNombre:(NSString*)nombre;
- (id)initWithNombre:(NSString*)nombre
                  descripcion:(NSString*)descripcion
                  horas:(NSUInteger)horas;

+ (id)asignatura;
+ (id)asignaturaWithNombre:(NSString*)nombre;
+ (id)asignaturaWithNombre:(NSString*)nombre
                  descripcion:(NSString*)descripcion
                  horas:(NSUInteger)horas;

+ (CGFloat)creditosParaHoras:(CGFloat)horas;
- (CGFloat)creditos;
- (CGFloat)tasaConPrecioPorCredito:(CGFloat)precioCredito
                                esBecario:(BOOL)becario;

@property(nonatomic,retain) NSString *nombre;
@property(nonatomic,retain) NSString *descripcion;
@property(nonatomic,assign) NSUInteger horas;

@end
```

En el anterior ejemplo hemos declarado tres propiedades (nombre, descripción y horas). Con esto realmente lo que estamos haciendo es declarar implícitamente que vamos a tener los métodos accesores siguientes:

```
- (NSString *)nombre;
- (void)setNombre: (NSString *)nombre;
- (NSString *)descripcion;
- (void)setDescripcion: (NSString *)descripcion;
- (NSUInteger)horas;
- (void)setHoras: (NSUInteger)horas;
```

Pero si sólo declaramos las propiedades en la interfaz obtendremos un *warning* del compilador, ya que no tenemos la implementación de los métodos implícitos que hemos declarado. Además, simplemente estamos declarando los métodos accesores, por el momento no los estamos vinculando a ninguna variable de instancia. Podemos hacer que la implementación se genere de forma automática incluyendo la etiqueta `@synthesize` seguida del nombre de la propiedad cuyos métodos accesores queremos implementar. Si el nombre de la propiedad no coincide con el nombre de la variable de instancia a la que vamos a acceder (como es el caso de nuestro ejemplo), podemos vincularla añadiendo `= nombreVariable` tras el nombre de la propiedad:

```
@implementation UAAsignatura

@synthesize nombre = _nombre;
@synthesize descripcion = _descripcion;
@synthesize horas = _horas;

// Implementacion del resto de metodos
...
@end
```

Con esto se crearán las implementaciones de los *getters* y *setters* sin tener que escribirlos de forma explícita en el fichero de código (el compilador ya sabe cómo crealos).

Nota

Es recomendable hacer que la propiedad tenga un nombre diferente al de la variable de instancia a la que se accede para evitar confusiones (un error común es utilizar la variable de instancia cuando se debería estar utilizando la propiedad). Habitualmente encontraremos que a las variables de instancia se le añade el prefijo `_` para evitar confusiones, como hemos hecho en nuestro ejemplo. Si las nombramos de forma distinta e intentamos utilizar la propiedad donde debería ir la variable de instancia (o viceversa) nos aparecerá un error de compilación, y pinchando sobre él en el entorno nos permitirá corregirlo automáticamente (*fix-it*).

Como alternativa a `@synthesize` también podríamos implementar manualmente los accesores para cada propiedad, de forma que podríamos personalizar la forma de acceder a los campos, o incluso introducir propiedades que no estén vinculadas a ninguna variable de instancia.

Nota

Si utilizamos `@synthesize` no es necesario declarar manualmente las variables de instancia asociadas a las propiedades. En caso de no existir las variables de instancia especificadas, el compilador se encargará de crearlas por nosotros.

5.1.1. Acceso a las propiedades

Las propiedades definen una serie de métodos accesores, por lo que podremos acceder a ellas a través de estos métodos, pero además encontramos una forma alternativa de acceder a ellas mediante el operador `..`. Por ejemplo podríamos utilizar el siguiente código:

```
asignatura.nombre = @"Plataforma iOS";
 NSLog(@"Nombre: %@", asignatura.nombre);
```

Este código es equivalente a llamar a los métodos accesores:

```
[asignatura setNombre: @"Plataforma iOS"];
 NSLog(@"Nombre: %@", [asignatura nombre]);
```

Advertencia

Hay que destacar que el operador `.` no sirve para acceder a las variables de instancia directamente, sino para acceder a las propiedades mediante llamadas a los métodos accesores. Este es uno de los motivos por los que es conveniente que las propiedades no tengan el mismo nombre que las variables de instancia, para evitar confundir ambas cosas.

5.1.2. Gestión de la memoria

Lo más habitual es que los objetos que referenciamos desde las variables de instancia de nuestros objetos deban ser retenidos para evitar que se libere su memoria durante la vida de nuestro objeto. Esto se tendrá que tener en cuenta en los *setters* generados para que se libere la referencia anterior (si la hubiese) y retenga la nueva. Esta es la política que se conoce como `retain`. En otros casos no nos interesa que se retenga, sino simplemente asignar la referencia (para así evitar posibles referencias cíclicas que puedan causar fugas de memoria). Esta política se conoce como `assign`. Por último, puede que nos interese no sólo retener el objeto, sino retener una copia del objeto que se ha recibido como parámetro para evitar efectos laterales al modificar un objeto mutable desde lugares diferentes. Esta política se conoce como `copy`, y sólo la podremos aplicar si el tipo de datos de la propiedad implementa el protocolo `NSCopying`.

La política de gestión de memoria se indica como atributo de la etiqueta `@property` (`assign`, `retain`, `copy`). Vamos a ver cada una de ellas con más detalle.

El caso más sencillo es el de la política `assign`. Esta será la política que se aplique por defecto, o la que se debe aplicar en cualquier tipo de datos que no sea un objeto, aunque

también podemos aplicarla a objetos. Los accesores generados con ella tendrán la siguiente estructura:

```
- (NSString *)nombre {
    return _nombre;
}

- (void)setNombre:(NSString *)nombre {
    _nombre = nombre;
}
```

La política más común para las propiedades de tipo objeto será `retain`. Con ella el *getter* será igual que en el caso anterior, mientras que el *setter* tendrá la siguiente estructura:

```
- (void)setNombre:(NSString *)nombre {
    if(_nombre != nombre) {
        [nombre retain];
        [_nombre release];
        _nombre = nombre;
    }
}
```

En este caso en el *setter* eliminamos la referencia anterior, si la hubiese (si no la hay, `_nombre` será `nil` y enviar el mensaje `release` no tendrá ningún efecto). El nuevo valor se retiene y se asigna a la variable de instancia.

Podemos ver también que se comprueba si el nuevo valor es el mismo que el anterior. En tal caso no se hace nada, ya que no merece la pena realizar tres operaciones innecesarias.

Por último, tenemos el modificador `copy`, que realiza una copia del objeto que pasamos como parámetro del *setter*. El *getter* será como en el caso anterior.

```
- (void)setNombre:(NSString *)nombre {
    NSString *nuevoObjeto = [nombre copy];
    [_nombre release];
    _nombre = nuevoObjeto;
}
```

En este caso el objeto que pasemos debe implementar el protocolo `NSCopying` para poder ser copiado. Siempre se realizará la copia inmutable (`copy`). Si queremos realizar una copia mutable deberemos definir nuestro propio *setter*. En este *setter* vemos también que no se comprueba si los objetos son el mismo, ya que en el caso de las copias es improbable (aunque podría pasar, ya que los objetos inmutables se copian simplemente reteniendo una referencia).

Tanto si se retiene (`retain`) como si se copia (`copy`) nuestro objeto tendrá una referencia pendiente de liberar. Se liberará cuando se asigne otro objeto a la propiedad, o bien cuando nuestro objeto sea destruido. En este último caso nosotros seremos los responsables de escribir el código para la liberación de la referencia en el método `dealloc`, ya que `@synthesize` sólo se ocupa de los *getters* y *setters*. En nuestro caso deberíamos implementar `dealloc` de la siguiente forma:

```
- (void)dealloc
{
    [_nombre release];
```

```
    [_descripcion release];
    [super dealloc];
}
```

Estamos liberando todas las variables de instancia asociadas a propiedades con política `retain` o `copy`. Nunca deberemos liberar propiedades con política `assign`.

De la misma forma, en el inicializador también deberíamos retener las variables de instancia si se les da un valor, ya que al utilizar estos modos se espera que las variables de instancia tengan una referencia pendiente de liberar cuando sean distintas de `nil`. Las variables que correspondan a propiedades de tipo `assign` nunca deberán ser retenidas, ya que eso provocaría una fuga de memoria cuando asignásemos un nuevo valor:

```
- (id)initWithNombre:(NSString*)nombre descripcion:(NSString*)descripcion
horas:(NSUInteger)horas
{
    self = [super init];
    if(self != nil) {
        _nombre = [nombre retain];
        _descripcion = [descripcion retain];
        _horas = horas;
    }
    return self;
}
```

Referencias cíclicas

Las referencias en las que se retiene el objeto (`retain`, `copy`) son conocidas como referencias fuertes, mientras que las de tipo `assign` son referencias débiles. Un problema que podemos tener con las referencias fuertes es el de las referencias cíclicas. Imaginemos una clase A con una referencia fuerte a B, y una clase B con una referencia fuerte a A. Puede que ninguna otra clase de nuestra aplicación tenga una referencia hacia ellas, por lo que estarán inaccesibles, pero si entre ellas tienen referencias su cuenta de referencias será mayor que 0 y por lo tanto nunca serán borradas (tenemos una fuga de memoria).

Para evitar las referencias cíclicas fundamentalmente debemos seguir la regla de que, dada una jerarquía de clases, sólo debe haber referencias fuertes desde las clases padre a las clases hijas. Las referencias que pueda haber desde una clase a sus ancestros deberán ser siempre débiles, para evitar así los ciclos. Si una clase A que ha sido referenciada débilmente desde otra clase B va a ser eliminada, siempre deberá avisarse a B de este hecho poniendo la referencia a `nil`, para evitar posibles accesos erróneos a memoria.

5.1.3. Propiedades atómicas

En los ejemplos anteriores hemos considerado siempre propiedades no atómicas (con el modificador `nonatomic`), que serán las que utilizaremos en casi todos los casos. Si no incluyésemos este modificador, la propiedades se considerarían atómicas, y el código de todos los `getters` y `setters` quedaría envuelto en un bloque sincronizado:

```
- (void)setNombre:(NSString *)nombre {
    @synchronized(self) {
        if(_nombre != nombre) {
```

```
        [nombre retain];
        [_nombre release];
        _nombre = nombre;
    }
}
```

Con la directiva `@synchronized` estamos creando un bloque de código sincronizado que utilizará como cerrojo el objeto actual (`self`). Esto es equivalente a declarar un método como `synchronized` en Java.

Pero donde encontramos una mayor diferencia es en los *getters* para los modos `retain` y `copy`. En las propiedades no atómicas estos métodos se limitaban a devolver el valor de la variable de instancia. Sin embargo, cuando la propiedad sea atómica tendrán el siguiente aspecto:

```
- (NSString *)nombre {
    @synchronized(self) {
        return [[_nombre retain] autorelease];
    }
}
```

¿Por qué se llama a `retain` y a `autorelease`? Esto nos va a garantizar que el objeto devuelto va a seguir estando disponible hasta terminar la pila de llamadas. Evitará que podamos tener problemas en casos como el siguiente:

```
NSString *oldNombre = asignatura.nombre;
asignatura.nombre = @"Nuevo nombre"; // Hace un release del nombre antiguo
NSLog(@"Antiguo nombre: %@", oldNombre);
```

Si no se hubiese retenido el nombre antiguo antes de devolverlo, el *setter* lo habría borrado de la memoria y podríamos tener un error de acceso en el `NSLog`. Con propiedades atómicas no tendremos problemas en este caso, pero con las no atómicas, que son las que utilizaremos más comúnmente este código daría error.

Como vemos, las propiedades atómicas, a parte de ser sincronizadas, incorporan dos operaciones adicionales de gestión de memoria en los *getters*, por lo que si se accede a ellas muy frecuentemente vamos a tener un impacto en el rendimiento. Por este motivo utilizaremos normalmente las propiedades no atómicas, aunque deberemos llevar cuidado para evitar problemas como el anterior.

5.1.4. Automatic Reference Counting (ARC)

A partir de Xcode 4.2 se incluye la característica *Automatic Reference Counting* (ARC), que nos permite dejar que sea el compilador quien se encargue de la gestión de la memoria, liberando al desarrollador de esta tarea. Es decir, ya no será necesario hacer ninguna llamada a los métodos `retain`, `release`, y `autorelease`, sino que el compilador se encargará de detectar dónde es necesario introducir las llamadas a dichos métodos y lo hará por nosotros.

Podemos activar ARC al crear un nuevo proyecto, o si ya tenemos un proyecto creado

podemos migrarlo mediante la opción *Edit > Refactor > Convert to Objective-C ARC*

Esta es una característica aportada por el compilador LLVM 3.0 de Apple. Por lo tanto, para comprobar si está activa deberemos ir a *Build Settings > Apple LLVM Compiler 3.0 - Language > Objective-C Automatic Reference Counting*.

Si decidimos trabajar con ARC, debemos seguir una serie de reglas:

- No sólo no es necesario utilizar `retain`, `release`, `autorelease`, sino que intentar llamar a cualquiera de estos métodos resultará en un error de compilación. Tampoco se permitirá consultar el contador de referencias (propiedad `retainCount`), ni deberemos llamar al método `dealloc`. Ya no será necesario por lo tanto implementar el método `dealloc` en nuestras clases, a no ser que queramos liberar memoria que no corresponda a objetos de Objective-C, o que queramos realizar cualquier otra tarea de finalización de recursos. En caso de implementar dicho método, no deberemos llamar a `[super dealloc]`, ya que esto nos daría un error de compilación. La cadena de llamadas al `dealloc` de la superclase está automatizada por el compilador.
- No debemos crear referencias a objetos Objective-C dentro de estructuras C, ya que quedaría fuera de la gestión que es capaz de hacer ARC. En lugar de estructuras C, podemos crear objetos Objective-C que hagan el papel de estructura de datos.
- No se debe crear un *autorelease pool* de forma programática como hemos visto anteriormente, sino que debemos utilizar un bloque de tipo `@autoreleasepool`:

```
@autoreleasepool {  
    ...  
}
```

- No se permite hacer un *cast* directo entre objetos de Objective-C (`id`) y punteros genéricos (`void *`).

ARC nos evita tener que realizar la gestión de la memoria, pero no nos protege frente a posibles retenciones cíclicas, las cuales causarían una fuga de memoria. Para evitar que esto ocurra deberemos marcar de forma adecuada las propiedades como fuertes o débiles, evitando siempre que se crucen dos referencias fuertes. Para esto se introducen dos nuevos tipos de propiedades:

- `strong`: Es equivalente a `retain`, aunque se recomienda pasar a usar `strong`, ya que en futuras versiones `retain` podría desaparecer (en las plantillas creadas veremos que siempre se utiliza `strong`, incluso cuando no está activado ARC). Las propiedades marcadas con `strong` retienen los objetos a los que referencian en memoria.
- `weak`: Referencia de forma débil a un objeto, es decir, sin retenerlo en memoria. Para que el objeto siga existiendo debe existir en algún otro lugar del código una referencia fuerte hacia él. Se diferencia de `assign` en que si el objeto es liberado, la propiedad `weak` pasará automáticamente a valer `nil`, lo cual nos protegerá de accesos inválidos a memoria.
- `unsafe_unretained`: Es equivalente a `assign`. Lo normal para objetos de Objective-C será utilizar siempre `strong` o `weak`, pero para tipos básicos deberemos utilizar `assign`. El tipo `unsafe_unretained` es equivalente y se podría utilizar

también en estos casos, pero por semántica, resultará más adecuado para referencias débiles a objetos que no se vayan a poner a `nil` de forma automática cuando se libere el objeto.

Atención

Sólo podremos utilizar propiedades `weak` en dispositivos iOS 5 (o superior) y en aplicaciones con ARC activo. Si queremos que nuestra aplicación sea compatible con versiones anteriores de iOS, podremos utilizar ARC (ya que las operaciones de gestión de memoria se añaden en tiempo de compilación), pero no podremos utilizar referencias de tipo `weak`. En su lugar deberemos utilizar el tipo `unsafe_unretained`, y asegurarnos de poner a `nil` el puntero de forma manual cuando el objeto vaya a ser liberado en otro lugar del código.

En el caso de las variables (no propiedades), por defecto siempre establecerán una referencia fuerte durante su tiempo de vida. Sin embargo, encontramos una serie de modificadores para cambiar este comportamiento:

- `__strong`: Es el tipo por defecto. No hace falta declararla explícitamente de esta forma, ya que cualquier variable por defecto establecerá una referencia fuerte con el objeto al que apunta.
- `__weak`: Funciona de la misma forma que las propiedades de tipo `weak`. Debemos llevar cuidado al utilizar este tipo, ya que si el objeto al que referenciamos no está referenciado de forma fuerte desde ningún otro sitio, será liberado al instante. Por ejemplo, en el siguiente código:

```
NSString __weak *cadena = [[NSString alloc]
                           initWithFormat:@"Edad: %d", edad];
NSLog(cadena);
```

Estaremos haciendo un *log* de `nil`, ya que la cadena recién creada no ha sido referenciada de forma fuerte por nadie, y por lo tanto ha sido liberada al instante.

- `__unsafe_unretained`: Funciona de la misma forma que las propiedades de tipo `unsafe_unretained`.
- `__autoreleasing`: Este tipo es útil cuando vayamos a pasar una variable por referencia, para evitar que el compilador se confunda con su ciclo de vida. Por ejemplo, se utilizará cuando pasemos un parámetro de error:

```
NSError __autoreleasing *error = nil;
[objetos realizarOperacionConError:&error];
```

El objeto `NSError` se instancia dentro del método al que llamamos, pero dentro de ese método se ve como una variable local, por lo que para el compilador su vida termina con la finalización del método, y por lo tanto nosotros no podríamos ver su contenido. Para evitar que esto ocurra, se utiliza el tipo `__autoreleasing` que añadirá el objeto al *autorelease pool* más cercano, y por lo tanto lo tendremos disponible en toda la pila de llamadas hasta llegar a la liberación del *pool*.

5.1.5. Métodos generados

Otros modificadores que podemos incorporar a las propiedades son `readonly` y

`readwrite`. Con ellas realmente lo que se está indicando es si sólo queremos declarar los *getters* (`readonly`), o si además también se quieren declarar los *setters* (`readwrite`). Por defecto se considerará que son de lectura/escritura.

Además, también podemos modificar el nombre de los *getters* y *setters* de una propiedad dada mediante los modificadores `getter=nombreGetter` y `setter=nombreSetter`.

5.1.6. Acceso directo a las variables de instancia

Si hemos declarado las variables de instancia como públicas, podremos acceder a ellas directamente con el operador `->`:

```
asignatura->_nombre = @"Plataforma iOS";
```

Hemos de tener en cuenta que si accedemos de esta forma no se estarán ejecutando los *getters* y *setters*, sino que estaremos manipulando directamente la variable, por lo que no se estará realizando ninguna gestión de las referencias al objeto.

Por este motivo se desaconseja totalmente esta forma de acceso. Siempre deberemos acceder a los campos de nuestros objetos con el operador `..`, o bien con los *getters* y *setters*.

Nota

El operador `.` no sólo se limita a las propiedades declaradas, sino que puede aplicarse a cualquier método definido en el objeto. Sin embargo, no debemos abusar de este posible uso, y deberíamos limitar la utilización de este operador al acceso a propiedades del objeto.

5.2. Colecciones de datos

En Objective-C encontramos distintos tipos de colecciones de datos, de forma similar al marco de colecciones de Java. Se trata de colecciones genéricas que pueden contener como datos cualquier objeto de Objective-C. Los tipos principales de colecciones que encontramos en Objective-C son `NSArray`, `NSSet`, y `NSDictionary`. De todos ellos podemos encontrar versiones tanto mutables como inmutables.

5.2.1. Wrappers de tipos básicos

Como hemos comentado, las colecciones pueden contener cualquier objeto de Objective-C, pero los tipos de datos básicos no son objetos (`int`, `float`, `char`, etc). ¿Qué ocurre si necesitamos crear una colección de elementos de estos tipos?

Para solucionar este problema encontramos objetos de Objective-C que se encargan de envolver datos de estos tipos en forma de objeto, para así poderlos incluir en colecciones. Estos objetos se conocen como *wrappers*.

El más sencillo es el que nos permite introducir un valor `nil` en la colección. Para ello

utilizaremos un objeto de tipo `NSNull`. Dado que el valor `nil` es único, no necesitamos más que una instancia de dicha clase. Por este motivo dicho objeto se define como *singleton*, y podremos obtenerlo de la siguiente forma:

```
[NSNull null]
```

Otro tipo de datos común son los valores numéricos (`BOOL`, `int`, `float`, etc). Todos estos tipos pueden representarse mediante la clase `NSNumber`:

```
NSNumber *booleano = [NSNumber numberWithBool: YES];
NSNumber *entero = [NSNumber numberWithInt: 10];
NSNumber *flotante = [NSNumber numberWithFloat: 2.5];
...
BOOL valorBool = [booleano boolValue];
int valorEntero = [entero intValue];
float valorFlotante = [flotante floatValue];
```

Por último, tenemos el caso de las estructuras de datos. Este caso ya no es tan sencillo, ya que podemos tener cualquier estructura definida por nosotros. La única forma de tener una forma genérica para encapsular cualquier estructura de datos es almacenar sus *bytes*. Para ello utilizaremos la clase `NSValue`.

```
typedef struct {
    int x;
    int y;
} Punto;
...
Punto p;
p.x = 1;
p.y = 5;
NSValue *valorPunto = [NSValue valueWithBytes:&p
                                         objCType:@encode(Punto)];
```

Podemos observar que al construir el objeto `NSValue` debemos proporcionar la dirección de memoria donde se aloja el dato que queremos almacenar, y además debemos indicar su tipo. Para indicar su tipo utilizamos la directiva `@encode`. Esta directiva toma como parámetro un tipo de datos (podría ser cualquier tipo básico, compuesto, o incluso objetos), y nos devuelve la representación de dicho tipo de datos que utiliza internamente Objective-C.

Si queremos recuperar el valor guardado, deberemos proporcionar una dirección de memoria con el espacio necesario para alojar dicho valor:

```
Punto punto;
[valorPunto getValue:&punto];
```

5.2.2. Listas

Las listas son colecciones en las que los datos se guardan en un orden determinado. Cada elemento se almacena en un índice de la lista. Las listas se definen mediante los tipos

`NSArray` (inmutable) y `NSMutableArray` (mutable).

En caso de la versión inmutable, deberemos inicializarlo a partir de sus elementos, ya que al ser inmutable no podremos añadirlos más adelante. En este caso normalmente utilizaremos un constructor que tome como parámetros los elementos de la lista. Este constructor recibe una lista de parámetros terminada en `nil`:

```
NSArray *lista = [NSArray arrayWithObjects: obj1, obj2, obj3, nil];
```

Es importante no olvidarnos de poner `nil` al final de la lista. Muchos métodos con número variable de parámetros se definen de esta forma. Podemos consultar la documentación para saber si debe llevar `nil` al final, o bien fijarnos en la firma del método que aparece en Xcode al autocompletar (si aparece ... `nil` se trata de una lista acabada en `nil`). Si no incluimos `nil` al final obtendremos un *warning* del compilador y un error en tiempo de ejecución.

Su versión mutable, `NSMutableArray`, incorpora inicializadores adicionales en los que se indica la capacidad inicial de la lista, aunque ésta podría crecer conforme añadamos datos:

```
NSMutableArray *listaMutable = [NSMutableArray arrayWithCapacity: 100];
```

5.2.2.1. Acceso a los elementos de la lista

Podemos saber el número de elementos que tiene una lista con el método `count`:

```
NSUInteger numElementos = [lista count];
```

Esto se aplicará a cualquier tipo de colección (no exclusivamente a las listas). En el caso de las listas, los índices irán desde 0 hasta `count-1`. Podemos acceder al objeto que esté en un índice determinado con `objectAtIndex`:

```
id primerObjeto = [lista objectAtIndex: 0];
```

También podemos buscar el índice en el que está un objeto determinado con `indexForObject`:

```
NSUInteger indice = [lista indexOfObject: obj];
if(indice==NSNotFound) {
    // Objeto no encontrado
}
```

El objeto proporcionado se comparará con cada objeto de la lista utilizando su método `isEqual`. Si lo que queremos es buscar la misma instancia, entonces deberemos utilizar `indexOfObjectIdenticalTo`.

5.2.2.2. Modificación de los elementos de la lista

En el caso de que nuestra lista sea mutable, podremos modificar la lista de elementos que contiene para añadir, insertar, reemplazar o eliminar elementos:

```
// A partir del indice 5 se mueven a la siguiente posición
[listaMutable insertObject:obj atIndex:5];

// Lo añade al final de la lista
[listaMutable addObject:obj];

// A partir del indice 5 se mueven a la anterior posición
[listaMutable removeObjectAtIndex:5];

// Es más eficiente, con coste constante
[listaMutable removeLastObject];

[listaMutable replaceObjectAtIndex:5 withObject:obj];
```

También podremos encontrar gran cantidad de variantes de los métodos anteriores.

5.2.3. Conjuntos

Un conjunto es una colección no ordenada de elementos distintos. Dos objetos iguales (según `isEqual`) no pueden repetirse dentro del conjunto. De la misma forma que las listas, existen en versión inmutable `NSSet` y mutable `NSMutableSet`.

Se puede inicializar de forma similar a las listas, según sea mutable o inmutable:

```
NSSet *conjunto = [NSSet setWithObjects: obj1, obj2, obj3, nil];
NSMutableSet *conjuntoMutable = [NSMutableSet setWithCapacity: 100];
```

Al igual que en las listas, contamos con el método `count` para conocer el número de elementos del conjunto, y con el método `containsObject` que nos dirá si el conjunto contiene un determinado objeto.

```
BOOL pertenece = [conjunto containsObject: obj];
```

En el caso de los conjuntos mutables, tenemos métodos para añadir o eliminar elementos:

```
// Solo lo añade si no pertenece todavía al conjunto
[conjuntoMutable addObject:obj];
[conjuntoMutable removeObject:obj];
```

Además de estas operaciones, también encontramos métodos para realizar las operaciones habituales sobre conjuntos (unión, intersección, resta, etc).

Existe otra subclase de `NSSet` a parte de `NSMutableSet`: `NSCountedSet`. La diferencia con las anteriores consiste en que en este caso cada objeto lleva asociado un contador que indica cuántas veces se encuentra repetido en el conjunto. Si añadimos varias veces el mismo objeto, lo que estaremos haciendo es incrementar su contador. Podemos obtener el número de veces que se ha añadido un objeto dado con `countForObject`. Este tipo de conjuntos también se denomina bolsa de objetos.

5.2.4. Diccionarios

Los diccionarios son un tipo de colección en la que cada objeto se encuentra asociado a una clave. Realmente lo que almacenan es un conjunto de pares *clave-valor*. Igual que en

los casos anteriores, tenemos un diccionario inmutable (`NSDictionary`) y uno mutable (`NSMutableDictionary`). Los diccionarios equivalen a la colección que en Java se denomina `Map`.

Al crear un diccionario ya no basta con dar una lista de objetos, sino que necesitaremos también una lista de claves:

```
NSDictionary *diccionario =  
    [NSDictionary dictionaryWithObjectsAndKeys:  
        obj1, @"clave1",  
        obj2, @"clave2",  
        obj3, @"clave3", nil];
```

Las claves pueden ser cualquier tipo de objeto, pero será conveniente que sean cadenas. Podemos también crear un diccionario mutable vacío:

```
NSMutableDictionary *diccionarioMutable =  
    [NSMutableDictionary dictionaryWithCapacity: 100];
```

No puede haber más de una ocurrencia de la misma clave (esto se comprobará con `isEqual`). Del diccionario podemos sacar el objeto asociado a una clave con el método `objectForKey`:

```
id obj = [diccionario objectForKey:@"clave1"];
```

Si no hay ningún objeto asociado a dicha clave, este método nos devolverá `nil`.

También podemos sacar la lista de todas las claves (`allKeys`) o de todos los objetos almacenados en el diccionario (`allValues`). Estas listas se obtendrán como un objeto del tipo `NSArray`:

```
NSArray *claves = [diccionario allKeys];  
NSArray *valores = [diccionario allValues];
```

En el caso de los diccionarios mutables tenemos también métodos para establecer el objeto asociado a una clave dada, o para borrarlo:

```
[diccionario setObject:obj forKey:@"clave1"];  
[diccionario removeObjectForKey:@"clave1"];
```

5.2.5. Recorrer las colecciones

La forma más sencilla de recorrer una lista es utilizar la estructura `for-in`.

```
for(id obj in lista) {  
    NSLog(@"Obtenido el objeto %@", obj);  
}
```

Si sabemos que todos los elementos de la lista son de un tipo concreto, podemos declarar los items con ese tipo. Por ejemplo, si tenemos una lista de cadenas podríamos recorrerlas con:

```
for(NSString *cadena in lista) {  
    NSLog(@"Obtenida la cadena %@", cadena);
```

```
}
```

Con esta estructura también podremos recorrer los elementos pertenecientes a un conjunto y las claves registradas en un diccionario. Si quisiésemos recorrer los objetos de un diccionario, sin tener que pasar por las claves, podríamos hacerlo de la siguiente forma:

```
for(id valor in [diccionario allValues]) {
    NSLog(@"Obtenido el objeto %@", valor);
}
```

Lo habitual será recorrer las claves, ya que a partir de ellas es muy sencillo obtener sus valores asociados:

```
for(NSString *clave in diccionario) {
    NSLog(@"%@", clave, [diccionario objectForKey: clave]);
}
```

También podemos utilizar objetos enumeradores (`NSEnumerator`) para recorrer las listas. Estos objetos vienen de versiones anteriores de Objective-C en las que no existía la estructura `for-in`. Actualmente es más sencillo y limpio utilizar dicho tipo de `for` para recorrer las colecciones.

```
NSEnumerator *enumerador = [lista objectEnumerator];
id obj;

while (obj = [enumerador nextObject]) {
    NSLog(@"Obtenido el objeto %@", valor);
}
```

Cuidado

Si mientras estamos recorriendo una colección mutable (tanto con `for-in` como con un enumerador) tratamos de modificarla, obtendremos un error en tiempo de ejecución.

5.2.6. Almacenamiento persistente

Las colecciones se pueden grabar en ficheros de forma muy sencilla. Para hacer esto todas ellas nos proporcionan un método `writeToFile:atomically:` que se encarga de volcar el contenido de la colección al fichero cuya ruta especificamos. A la hora de trabajar con ficheros, no podremos guardarlos en cualquier ruta, sino que deberemos utilizar las rutas que el dispositivo reserva a nuestra aplicación. Un directorio habitual donde podemos almacenar los datos es el directorio de documentos (representado por `NSDocumentDirectory`). Podemos obtener la ruta de dicho directorio de la siguiente forma:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                    NSUserDomainMask,
                                                    YES);

if ([paths count] > 0)
{
```

```
NSString *directory = [paths objectAtIndex:0];
NSString *filename = [directory
    stringByAppendingPathComponent:@"colección.plist"];
BOOL guardado = [colección writeToFile:filename
    atomically:YES];
}
```

Nota

El contenido el directorio de documentos será guardado automáticamente en las copias de seguridad que se realicen mediante iTunes o iCloud. Según los datos que almacenemos, deberemos elegir el directorio apropiado de nuestra aplicación. Para más información sobre el sistema de archivos se puede consultar el tutorial *File System Basics* de la documentación proporcionada por Apple.

La colección se almacena en un fichero con formato `plist` (hemos visto que gran parte de los ficheros de configuración del proyecto iOS utilizan este mismo formato). Debido a las limitaciones del formato, debemos tener en cuenta que sólo podremos guardar colecciones que contengan únicamente los siguientes tipos de datos:

- `NSString`
- `NSData`
- `NSDate`
- `NSNumber`
- `NSArray`
- `NSDictionary`

Si alguno de los componentes de la colección fuese de cualquier otro tipo la colección no se guardaría (la llamada a `writeToFile:atomically:` devolvería `NO`).

También podemos leer los datos de una colección a partir de un fichero `plist`. Para ello contamos con un inicializador, y su correspondiente método factoría:

```
- (id)initWithContentsOfFile:
+ (id)arrayWithContentsOfFile:
```

En este caso, podemos utilizarlo para leer los ficheros que hayamos almacenado anteriormente en el directorio de documentos:

```
BOOL fileExists = [[NSFileManager defaultManager]
    fileExistsAtPath:filename];
if (fileExists) {
    colección = [NSArray arrayWithContentsOfFile:filename];
}
```

Pero también podríamos leer ficheros `plist` que hayamos creado con Xcode y hayamos empaquetado junto a la aplicación (estos ficheros serán de sólo lectura):

```
NSString *filename = [[NSBundle mainBundle] pathForResource:@"datos"
    ofType:@"plist"];
colección = [NSArray alloc arrayWithContentsOfFile:filename];
```

Si queremos almacenar objetos propios que no pertenezcan a la lista de tipos de datos mencionada anteriormente, podemos hacerlo adoptando el protocolo NSCoding:

```
@interface UAAsignatura : NSObject<NSCoding>
...
@end
```

Este protocolo nos obliga a definir dos métodos que se encargarán de serializar y deserializar el objeto:

```
- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super init];
    if(self!=nil) {
        self.nombre = [aDecoder decodeObjectForKey:@"nombre"];
        self.descripcion = [aDecoder decodeObjectForKey:@"descripcion"];
        self.horas = [aDecoder decodeIntegerForKey:@"horas"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:self.nombre forKey:@"nombre"];
    [aCoder encodeObject:self.descripcion forKey:@"descripcion"];
    [aCoder encodeInteger:self.horas forKey:@"horas"];
}
```

Una vez definidos estos métodos, podremos guardar o leer el objeto de forma sencilla con NSKeyedArchiver y NSKeyedUnarchiver:

```
+ (UAAsignatura *) load {
    UAAsignatura *asig = nil;

    NSArray *dirs = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    if([dirs count] > 0) {
        NSString *filename = [[dirs objectAtIndex:0]
            stringByAppendingPathComponent: @"datos"];
        asig = [NSKeyedUnarchiver unarchiveObjectFromFile: filename];
    }

    return asig;
}

- (BOOL) save {
    NSArray *dirs = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    if([dirs count] > 0) {
        NSString *filename = [[dirs objectAtIndex:0]
            stringByAppendingPathComponent: @"datos"];

        return [NSKeyedArchiver archiveRootObject: self
            toFile: filename];
    } else {
        return NO;
    }
}
```

Nota

Debemos destacar que todas las colecciones implementan NSCoding, por lo que podemos almacenarlas también de la forma anterior.

5.3. Key-Value-Coding (KVC)

KVC es una característica del lenguaje Objective-C que nos permite acceder a las propiedades de los objetos proporcionando una cadena con el nombre de la propiedad. Todos los objetos incorporan un método `valueForKey:` que nos permite acceder a las propiedades de esta forma:

```
NSString *nombre = [asignatura valueForKey: @"nombre"];
NSNumber *horas = [asignatura valueForKey: @"horas"];
```

Esto será útil cuando no conocemos el nombre de la propiedad a la que queremos acceder hasta llegar a tiempo de ejecución. Podemos observar también que cuando tenemos propiedades que no son objetos, las convierte al *wrapper* adecuado (en el ejemplo anterior `NSUInteger` a pasado a `NSNumber`).

También podemos modificar el valor de las propiedades mediante KVC con el método `setValue:forKey:`:

```
[asignatura setValue:@"Proyecto iOS" forKey:@"nombre"];
[asignatura setValue:[NSNumber numberWithInteger:30] forKey:@"horas"];
```

Debemos destacar que KVC siempre intentará acceder a las variables de instancia a través de los métodos accesores (para que los encuentre deberán llamarse de la forma estándar: `nombre` y `setNombre`), pero si estos métodos no estuviesen disponibles, accedería a ella directamente de la misma forma, incluso tratándose de una variable privada. El modificador `@private` sólo afecta al compilador, pero no tiene ningún efecto en tiempo de ejecución.

También podemos acceder a propiedades mediante una ruta de claves (*key path*). Esto será útil cuando tengamos objetos anidados y desde un nivel alto queramos consultar propiedades de objetos más profundos en la jerarquía. Para ello tenemos `valueForKeyPath:` y `setValue:forKeyPath:`:

```
NSString *nombreCoordinador =
[asignatura valueForKeyPath:@"coordinador.nombre"];
```

Cada elemento de la ruta se separará mediando el símbolo punto (.) .

5.3.1. KVC y colecciones

Cuando aplicamos KVC sobre colecciones de tipo `NSArray` o `NSSet`, lo que estaremos haciendo es acceder simultáneamente a todos los elementos de la colección. Esto nos permite modificar una determinada propiedad de todos los elementos con una única operación, u obtener la lista de valores de una determinada propiedad de los objetos de la colección.

```
[lista setValue:[NSNumber numberWithInteger:50] forKey:@"horas"];
NSArray *nombres = [lista valueForKey:@"nombre"];
```

```

for(NSString *nombre in nombres) {
    NSLog(@"Nombre asignatura: %@", nombre);
}

```

El caso de `NSDictionary` es distinto. De hecho, la forma de acceder a las propiedades de los objetos mediante KVC (`valueForKey:`) nos recuerda mucho a la forma en la que se accede a los objetos en los diccionarios a partir de su clave (`objectForKey:`). La diferencia entre ambos métodos es que en KVC las claves siempre deben ser cadenas, mientras que en un diccionario podemos utilizar como clave cualquier tipo de objeto. Sin embargo, si tenemos un diccionario en el que las claves sean cadenas, también podremos acceder a los objetos mediante el método `valueForKey:` de KVC. Por este motivo se recomienda utilizar siempre cadenas como claves.

5.3.2. Acceso mediante listas

Puede que algunas variables de instancia de nuestros objetos sean colecciones de datos, y podremos acceder a ellas y a sus elementos como hemos visto anteriormente. Pero puede que tengamos alguna propiedad que nos interese que se comporte como un *array*, pero que realmente no esté asociada a una variable de instancia de tipo `NSArray`.

Cuando teníamos propiedades con un único valor (por ejemplo `nombre`), nos bastaba con tener definidos dos métodos accesores: el *getter* con el mismo nombre que la propiedad (`nombre`) y el *setter* con el prefijo `set` (`setNombre`), aunque no existiese realmente ninguna variable de instancia llamada `nombre`.

En el caso de las listas necesitaremos definir métodos adicionales. Por ejemplo, vamos a considerar que tenemos una propiedad `profesores`, pero no tenemos a los profesores almacenados en ningún `NSArray`. Podemos hacer que KVC acceda a ellos como si se tratase de una lista definiendo los siguientes métodos:

```

- (NSUInteger) countOfProfesores {
    return numeroProfesores;
}

- (id) objectInProfesoresAtIndex: (NSUInteger) index {
    [self obtenerProfesorEnPosicion: index];
}

```

Si definimos estos métodos, aunque los profesores no estén almacenados en ningún *array*, podremos obtenerlos en forma de `NSArray` mediante KVC:

```
NSArray *profesores = [asignatura valueForKey: @"profesores"];
```

Podríamos también obtener el listado de una determinada propiedad de los profesores, utilizando un *key path*, o modificar la misma propiedad para todos los profesores.

Esta característica resultará de gran utilidad para el acceso a bases de datos, ya que nos permitirá de forma sencilla definir objetos que mapeen los datos almacenados en la BD a objetos.

Podemos incluso acceder al objeto mediante un *array* de tipo mutable. Para ello necesitamos implementar dos métodos adicionales:

```
- (void) insertObject:(id)obj inProfesoresAtIndex:(NSUInteger)index {
    [self insertarProfesor: obj enPosicion: index];
}

- (void) removeObjectFromProfesoresAtIndex: (NSUInteger) index {
    [self eliminarProfesorEnPosicion: index];
}
```

Si añadimos estos métodos podremos obtener un `NSMutableArray` para manipular los profesores, aunque internamente estén almacenados de otra forma:

```
NSMutableArray *profesores =
    [asignatura mutableArrayValueForKey: @"profesores"];
```

5.4. Programación de eventos

Hemos visto que en Objective-C no existe sobrecarga de métodos. Los métodos se identifican por lo que se conoce como el *selector*, que consiste en el nombre del método seguido de los nombres de sus parámetros. Por ejemplo `setValue:forKey:`. Cada parámetro se indica mediante : en el selector, y no importa de qué tipo sea (por eso no existe sobrecarga).

Los selectores se puede representar en el código mediante el tipo `SEL`, y podemos crear valores de este tipo mediante la directiva `@selector()`:

```
SEL accesoKVC = @selector(setValue:forKey:);
```

Podremos utilizar esta variable de tipo `SEL` para ejecutar el *selector* indicado sobre cualquier objeto, utilizando el método `performSelector:` (o cualquiera de sus variantes) del objeto sobre el que lo queramos ejecutar.

```
[asignatura performSelector:accesoKVC
    withObject:@"Plataforma iOS"
    withObject:@"nombre"];
```

Esta forma de ejecutar métodos va a ser de gran utilidad para definir *callbacks*. Cuando queramos que se nos notifique de un determinado evento en el momento que se produzca, podemos proporcionar una referencia a nuestro objeto (al que nos referiremos como *target*) y el *selector* al que queremos que se nos avise.

5.4.1. Patrón target-selector

Como hemos comentado, la forma anterior de ejecutar selectores va a permitirnos definir *callbacks* de forma muy sencilla. Por ejemplo, la clase `NSTimer` utiliza este esquema. En la mayoría de sus inicializadores toma como parámetros:

```
NSTimer *temporizador = [NSTimer
    scheduledTimerWithTimeInterval:5.0
```

```
target: self
selector: @selector(tick:)
userInfo: nil
repeats: YES];
```

De esta forma el temporizador llamará cada 5 segundos al método `tick` de nuestro objeto (`self`). Cuando utilizamos este esquema, la firma del método al que se llama está determinada por quien genera los eventos. Normalmente, este tipo de métodos recibirán un parámetro con una referencia al objeto que produjo el evento. En el caso del temporizador anterior el método deberá tomar un parámetro de tipo `NSTimer*`, en el que recibiremos el objeto temporizador que está generando los eventos.

```
- (void) tick: (NSTimer*)temporizador;
```

Normalmente el objeto especificado como `target` nunca se retendrá. Esto es así porque es muy frecuente que la clase que se registra como `target` sea ancestra de la que genera los eventos. Por ejemplo, en el caso del temporizador, lo más común es que la clase que crea el temporizador y lo retiene en una de sus variables de instancia sea también la clase que se registra como destino de los eventos del temporizador (especificando `target: self` al crear el temporizador). Si el temporizador retuviese el `target` tendríamos una retención cíclica y por lo tanto una posible fuga de memoria.

Por lo tanto, cuando utilicemos este patrón para definir un *callback*, deberemos llevar cuidado cuando el objeto desaparezca. Antes siempre deberemos eliminarlo de todos los lugares en los que lo hubiésemos establecido como `target` (a no ser que el `target` se haya definido como una propiedad de tipo `weak`, con las que esto se haría de forma automática).

5.4.2. Notificaciones

Otra forma de informar de que un evento ha ocurrido es mediante el uso de notificaciones. Las notificaciones son una especie de mensajes de tipo *broadcast* que podemos enviar, o bien escuchar los que otros envían. Una diferencia entre las notificaciones y otros mecanismos de comunicación es que las notificaciones se envían sin saber quién va a ser el receptor, podría incluso no haber ningún receptor o haber varios de ellos. Esto nos va a permitir comunicar objetos lejanos en el diagrama de clases, que muchas veces resultan difícilmente accesibles el uno desde el otro.

La gestión de las notificaciones se hace mediante el *notification center*, que se define como *singleton*:

```
[NSNotificationCenter defaultCenter]
```

A través de este centro de notificaciones podemos difundir objetos de tipo `NSNotification`. Estos objetos se componen de tres elementos básicos:

- `name`: Nombre de la notificación que sirve para identificarla.
- `object`: Objeto que se adjunta a la notificación. Suele ser el objeto que la envía.
- `userInfo`: Información adicional que queramos añadir, en forma de diccionario

`NSDictionary`). La información aquí incluida dependerá del tipo de notificación, deberemos consultar la documentación de cada una para obtener esta información.

Podemos crear una notificación de la siguiente forma:

```
NSNotification *notificacion = [NSNotification  
    notificationWithName:@"SincronizacionCompletada"  
    object:self  
    userInfo:nil];
```

Existen otros métodos factoría que toman menos parámetros, por ejemplo, en muchas ocasiones no necesitamos proporcionar ninguna información en `userInfo`. En esos casos podemos utilizar un método factoría más sencillo sin este parámetro.

Una vez tenemos la notificación, podemos enviarla a través del centro de notificaciones.

```
[[NSNotificationCenter defaultCenter] postNotification: notificacion];
```

En lugar de crear el objeto `NSNotification` previamente, existe un atajo para que se cree y se envíe con una única operación:

```
[[NSNotificationCenter defaultCenter]  
    postNotificationName:@"SincronizacionCompletada" object:self];
```

Por otro lado, si lo que queremos es escuchar las posibles notificaciones de un determinado tipo (nombre) que se produzcan en la aplicación, tendremos que registrar un observador en el centro de notificaciones.

```
[[NSNotificationCenter defaultCenter]  
    addObserver:self  
    selector:@selector(sincronizado:)  
    name:@"SincronizacionCompletada"  
    object:nil];
```

Podemos ver que para registrarnos como observador estamos utilizando el patrón *target-selector* visto anteriormente. Cada vez que se produzca una notificación de tipo @"SincronizacionCompleta", se enviará un mensaje al método `sincronizado:` de nuestro objeto (`self`). Este método deberá recibir como parámetro un objeto de tipo `NSNotification`, con el que recibirá la notificación cuando se produzca.

```
- (void)sincronizado:(NSNotification *) notificacion {  
    NSLog(@"Sincronizacion completada");  
}
```

El último parámetro nos permite especificar si queremos que sólo nos llegan notificaciones enviadas por un objeto concreto. Si queremos cualquier notificación del tipo indicado, independientemente de su fuente, especificaremos `object:nil`.

Podemos dejar de ser observador con el siguiente método:

```
[[NSNotificationCenter defaultCenter] removeObserver: self];
```

Con esto nos elimina como observador de cualquier notificación. Si queremos eliminar sólo una notificación concreta debemos usar otra versión de este método

```
removeObserver:name:object:).
```

Importante

Siempre deberemos llamar a `removeObserver:` antes de que nuestro objeto sea desalojado de la memoria, ya que el centro de notificaciones no retiene a los observadores, siguiendo las normas que hemos comentado anteriormente para el patrón *target-selector*.

Los nombres de notificaciones se suelen definir como constantes. En el ejemplo hemos creado un tipo propio de notificación, pero si utilizamos notificaciones incorporadas en la API de Cocoa Touch siempre deberemos utilizar las constantes que se definen para ellas, y nunca la cadena de texto que corresponde a dichas constantes. Si en un momento dado cambiase la cadena de texto, o bien nos equivocásemos al escribir algún carácter, dejaríamos de recibir las notificaciones correctamente.

5.4.3. Key Value Observing (KVO)

KVO nos permitirá observar los posibles cambios que se produzcan en las propiedades de los objetos. Recordemos que KVC nos permitía acceder a dichas propiedades de forma dinámica mediante cadenas de texto. KVO se podrá utilizar en cualquier propiedad que implemente KVC. Para registrarnos como observador de cualquier objeto, tenemos el siguiente método:

```
[_asignatura addObserver:self
    forKeyPath:@"nombre"
        options:NSMutableArrayObservingOptionNew
        context:NULL];
```

Cuando la propiedad `nombre` del objeto `_asignatura` cambie de valor, recibiremos el siguiente mensaje en nuestro objeto (`self`):

```
observeValueForKeyPath:ofObject:change:context:
```

Por lo tanto, deberemos implementar el correspondiente método en nuestro observador:

```
- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
        change:(NSDictionary *)change
        context:(void *)context
```

En `keyPath` recibiremos la ruta de la propiedad que ha cambiado, en `object` la referencia al objeto observado en el que se produjo el cambio, en `change` el valor que ha cambiado, y por último en `context` nos llegará la misma información que indicamos en dicho parámetro cuando nos registramos como observador.

Destacamos que el valor de la propiedad que ha cambiado lo recibimos como diccionario en el parámetro `change`. Esto es así porque podremos recibir varios valores (por ejemplo, el valor antiguo anterior al cambio, y el nuevo). Esto dependerá de lo que indicásemos en el parámetro `options` cuando nos registramos como observador. En el ejemplo anterior sólo hemos solicitado obtener el valor nuevo. Podremos obtener dicho valor a partir del

diccionario de la siguiente forma:

```
id valor = [change objectForKey: NSKeyValueChangeNewKey];
```

En este caso hemos utilizado también el patrón observador, como en los casos anteriores en los que hemos utilizado *target-selector*. La única diferencia de este último caso es que no nos ha permitido especificar el selector, como ocurría en los casos anteriores, sino que éste ya está establecido. Igual que en los casos anteriores, el objeto observado no retendrá al observador, por lo que es importante que eliminemos el observador antes de que se libere de la memoria. Para ello utilizaremos `removeObserver:forKeyPath:`.

5.4.4. Objetos delegados y protocolos

Otra forma habitual de dar respuesta a eventos es sobrescribir una serie de métodos a los que esperamos que el sistema llame cada vez que se produzca un evento. Sin embargo, muchas veces no es conveniente heredar de determinadas clases de la API (especialmente no contando con herencia múltiple) cuando lo único que nos interesa es dar respuesta a una serie de eventos que se pueden producir.

Podemos resolver esto mediante el patrón de **objetos delegados**, que nos permite que una determinada clase delegue en una clase secundaria para determinadas tareas. Este patrón se lleva a cabo mediante la inclusión de una propiedad `delegate`. Si dicha propiedad es distinta de `nil`, cuando ocurran determinados eventos pasará mensajes a `delegate` para que dicho objeto delegado realice las tareas oportunas.

Un ejemplo de uso de objetos delegados lo tenemos en la clase `UIApplication`. Para controlar el ciclo de vida de las aplicaciones no estamos creando una subclase de `UIApplication`, sino que pasamos dicha tarea a un objeto delegado.

Los objetos delegados suelen implementar un determinado protocolo, que les forzará a definir una serie de métodos que corresponderán a los mensajes que se espera que envíe el objeto principal. Este es el uso principal de los protocolos. Por ejemplo, en el caso del delegado de `UIApplication`, vemos que implementa el protocolo `UIApplicationDelegate` que incorpora una serie de métodos para controlar el ciclo de vida de la aplicación.

Podemos verlo como algo similar a los *listeners* en Java, pero con algunas diferencias. En el caso de los delegados, muchos métodos definidos en los protocolos son de implementación opcional, e incluso en algunas ocasiones puede que el delegado no se defina con un protocolo, sino que simplemente se documente qué métodos se espera que nos envíen. Esto es posible gracias a que Objective-C permite pasar cualquier mensaje a los objetos, aunque en su tipo no esté declarado el método.

¿Cómo podemos entonces saber en el objeto principal si el método al que le vamos a pasar el mensaje existe en el delegado? Si enviamos un mensaje a `nil` no pasa nada (obtenemos `nil`), pero si enviamos un mensaje a un objeto distinto de `nil` que no implementa ningún método para dicho mensaje, obtendremos una excepción, con lo que

deberemos llevar cuidado con esto. Para poder comprobar si el método está implementado o no, podemos utilizar los mecanismos de introspección que veremos a continuación.

5.4.5. Bloques

Los bloques son una nueva característica del lenguaje incorporada a partir de iOS 4.0. Nos permiten definir un bloque de código que se podrá pasar como parámetro para que se ejecute cuando un determinado evento suceda, o bien devolver un bloque como resultado de la ejecución de un método. Los bloques se declaran mediante el símbolo ^:

```
int (^Suma)(int, int) = ^(int num1, int num2) {
    return num1 + num2;
};
```

Dicho bloque tendrá como nombre `Suma` y recibirá dos parámetros numéricos. Podremos ejecutarlo como si se tratase de una función, pero realmente se está definiendo como una variable que puede ser pasada como parámetro o devuelta como resultado:

```
int resultado = Suma(2, 3);
```

Los bloques son una forma simple de definir *callbacks*. La ventaja de los bloques es que el código del *callback* se puede definir en el mismo lugar en el que se registra, sin necesitar crear un método independiente. Por ejemplo podemos registrarnos como observadores para una notificación usando bloques de la siguiente forma:

```
- (id)addObserverForName:(NSString *)name
    object:(id)obj
    queue:(NSOperationQueue *)queue
usingBlock:(void (^)(NSNotification *))block
```

En este caso vemos que como último parámetro define un bloque que tomará un parámetro de tipo `NSNotification *` y devuelve `void`. Lo podríamos utilizar de la siguiente forma:

```
[[NSNotificationCenter defaultCenter]
    addObserverForName:@"SincronizacionCompletada"
    object:nil
    queue:nil
usingBlock:^(NSNotification *notificacion) {
    NSLog(@"Se ha completado la sincronizacion");
}];
```

5.5. Introspección

En muchas ocasiones en nuestro código podemos recibir objetos con referencias de tipo `id`, en los que puede que no estemos seguros del tipo de objeto del que se trate, ni de los *selectores* que incorpora, o incluso los objetos pueden implementar protocolos que definan métodos opcionales, que puedan no estar presentes.

Por este motivo son necesarios mecanismos de introspección, que nos permitan en tiempo

de ejecución determinar de qué tipo es un objeto dado y si responde ante un determinado *selector*.

5.5.1. Tipo de la clase

Para conocer si un objeto es de una determinada clase podemos utilizar los siguientes métodos:

```
[asignatura isKindOfClass:[UAAsignatura class]]; // YES
[asignatura isKindOfClass:[NSObject class]]; // NO
[asignatura isKindOfClass:[UAAsignatura class]]; // YES
[asignatura isKindOfClass:[NSObject class]]; // YES
```

Podemos ver que `isKindOfClass` comprueba si una clase es estrictamente del tipo especificado, mientras que `isKindOfClass` indica si son tipos compatibles (es decir, si es la misma clase, o una subclase de la indicada). También podemos ver que podemos obtener el objeto que representa una clase dada mediante el método de clase `+class` que podremos encontrar en todas las clases. Nos devolverá un objeto de tipo `Class` con información de la clase.

También podemos saber si un objeto implementa un protocolo determinado con `conformsToProtocol:`. En este caso necesitaremos una variable de tipo `Protocol`, que podemos obtener mediante la directiva `@protocol`:

```
[asignatura conformsToProtocol:@protocol(NSCopying)];
```

5.5.2. Comprobación de selectores

Pero incluso hay protocolos que definen métodos opcionales, por lo que la forma más segura de saber si un objeto implementa un determinado método es comprobarlo mediante `respondsToSelector:`.

```
[asignatura respondsToSelector:@selector(creditos)];
```

Esta comprobación será muy común cuando utilicemos el patrón de objetos delegados, para comprobar si el delegado implementa un determinado método.

En las clases podemos encontrar un método similar llamado `instancesRespondToSelector:`, que indica si las instancias de dicha clase responden al *selector* indicado.

También podemos consultar la firma (*signature*) de un método a partir de su *selector*:

```
NSMethodSignature *firma = [asignatura methodSignatureForSelector:
                            @selector(tasaConPrecioPorCredito:esBecario:)];
```

Al igual que existe una variable implícita `self` que nos da un puntero al objeto en el que estamos actualmente, cuando estamos ejecutando un método disponemos también de otra variable implícita de nombre `_cmd` y de tipo `SEL` que indica el selector en el que estamos.

5.5.3. Llamar a métodos mediante implementaciones

A partir del *selector* de un método podemos obtener su implementación, de tipo `IMP`. La implementación es la dirección de memoria donde realmente se encuentra el método a llamar. Podemos utilizar la implementación para llamar al método como si se tratase de una función C. Esto nos permitirá optimizar las llamadas, ya que no será necesario resolver la dirección a la que llamar cada vez que se ejecuta de esta forma, se resolverá una única vez al obtener la implementación, y a partir de ese momento todas las llamadas que hagamos ya conocerán la dirección a la que llamar.

```
SEL selDescripcion = @selector(description);
IMP descripcion = [asignatura methodForSelector:selDescripcion];
NSString *descripcion = descripcion(asignatura, selDescripcion);
```

Vemos que en la llamada a la implementación debemos pasarle al menos dos parámetros, que corresponden a las dos variables implícitas con las que siempre contaremos en los métodos de nuestros objetos: `self` y `_cmd`. De hecho, el tipo `IMP` se define de la siguiente forma:

```
typedef id (*IMP)(id, SEL, ...);
```

Esto nos servirá siempre que el método al que queramos llamar devuelva un objeto (`id`). Si no fuese así, tendríamos que crear nuestro propio tipo de función. Por ejemplo, para `(CGFloat) tasaConPrecioPorCredito:(CGFloat)precioCredito esBecario:(BOOL)becario;` podríamos definir el siguiente tipo de función:

```
typedef CGFloat (*UACalculaTasa)(id, SEL, CGFloat, BOOL);
```

A la hora de obtener la implementación habrá que hacer una conversión *cast* al tipo de función al que se ajusta nuestro método:

```
SEL selTasa = @selector(tasaConPrecioPorCredito:esBecario:);
UACalculaTasa calcularTasa =
    (UACalculaTasa)[asignatura methodForSelector:selTasa];
CGFloat tasa = calcularTasa(asignatura, selTasa, 50, NO);
```

Podemos también obtener la implementación de un método de instancia a partir de un objeto de tipo `Class` mediante el método `instanceMethodForSelector:`.

5.5.4. Reenvío de mensajes

Cuando un objeto reciba un mensaje para el cual no tenga ningún método definido, realmente lo que ocurrirá es que se llamará a su método `forward::`, que recibirá el *selector* y la lista de parámetros que se han recibido. Por defecto este método lo que hará será lanzar una excepción indicando que el objeto no reconoce el *selector* solicitado y se interrumpirá la aplicación.

El método `forward::` de `NSObject` es privado y no debe ser sobrescrito, pero si que

podemos sobrescribir `forwardInvocation:` para modificar este comportamiento. Podríamos por ejemplo reenviar los mensajes a un objeto delegado, o bien simplemente ignorarlos.

Podemos aprovechar esta característica para crear accesores a propiedades en tiempo de ejecución. En ese caso, para evitar que el compilador nos dé un *warning* por no haber utilizado `@synthesize` para generar dichos accesores, utilizaremos la directiva `@dynamic` para indicar al compilador que no debe preocuparse por la propiedad indicada, y que puede confiar en que hemos implementado algún mecanismo para acceder a ella.

5.5.5. Objetos y estructuras

Objective-C nos permite convertir de forma sencilla un objeto a una estructura de datos. Para ello tenemos la directiva `@defs()`, que genera en tiempo de compilación la lista de variables de instancia de una clase. De esta forma, podemos generar una estructura de datos equivalente:

```
typedef struct {
    @defs(UAAsignatura);
} UAAsignaturaStruct;
```

Si tenemos un objeto de tipo `UAAsignatura`, podemos asignarlo directamente a una estructura equivalente mediante un *cast*:

```
UAAsignatura *asignatura = [[UAAsignatura alloc] init];
UAAsignaturaStruct *asigStruct = (UAAsignaturaStruct *) asignatura;
asigStruct->nombre = @"Plataforma iOS";
```

Advertencia

No debemos hacer esto si utilizamos ARC, ya que en este caso las conversiones directas entre `id` y `void *` están prohibidas.

5.6. Ciclo de vida de las aplicaciones

Ya hemos visto el patrón delegado, que se utiliza frecuentemente en Cocoa Touch. El primer uso que le daremos será para definir el delegado de la aplicación. La aplicación se implementa en la clase `UIApplication`, pero para poder tratar los eventos de su ciclo de vida necesitaremos crear un delegado en el que programaremos las tareas a realizar para cada uno de ellos. Este delegado implementará el protocolo `UIApplicationDelegate`, que define los métodos que podemos implementar para tratar cada uno de los eventos del ciclo de vida de la aplicación.

El evento fundamental es `application:didFinishLaunchingWithOptions:`, que se ejecutará cuando la aplicación ha terminado de cargarse y se va a poner en marcha. Aquí tendremos que programar las tareas a realizar para poner en marcha nuestra aplicación. Una implementación típica consiste en configurar el contenido a mostrar en la ventana de

la aplicación y mostrarla en pantalla:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];

    return YES;
}
```

En la próxima sesión veremos con más detalle como trabajar con los objetos de la interfaz.

El método opuesto al anterior es `applicationWillTerminate:`. Con él se nos notifica que la aplicación va a ser cerrada, por lo que deberemos liberar los objetos de memoria, parar las tareas en curso, y guardar los datos pendientes.

Durante la ejecución de la aplicación es posible que suceda algún evento externo, como por ejemplo una llamada entrante, que haga que nuestra aplicación se detenga temporalmente, para luego continuar por donde se había quedado (en el ejemplo anterior se reanudaría al terminar la llamada). Los eventos que nos notificarán la pausa y reanudación de la aplicación son `applicationWillResignActive:` y `applicationDidBecomeActive:` respectivamente.

A partir de iOS 4.0 aparece la multitarea, permitiéndonos dejar una aplicación en segundo plano sin finalizar su ejecución. Esto es diferente al caso anterior, ya que siempre se ha permitido que la aplicación entre en pausa temporalmente cuando suceden determinados eventos del dispositivo (como las llamadas entrantes), pero con la multitarea podemos dejar la aplicación en segundo plano para ejecutar cualquier otra aplicación, y posteriormente poder volver a recuperar el estado anterior de nuestra aplicación. Para implementar la multitarea deberemos definir los métodos `applicationDidEnterBackground:` y `applicationWillEnterForeground:`.

Deshabilitar la multitarea

Por defecto, cuando salimos de una aplicación realmente la estaremos dejando en segundo plano. La aplicación sólo se finalizará cuando la cerramos explícitamente desde la lista de aplicaciones recientes, o cuando el dispositivo la cierre por falta de recursos. Sin embargo, en algunos casos nos puede interesar deshabilitar la multitarea para que al salir de la aplicación, ésta se finalice. Para que esto sea así, añadiremos la propiedad "UIApplicationExitsOnSuspend" = YES en Info.plist.

Otro evento que resulta recomendable tratar es `applicationDidReceiveMemoryWarning:`. Recibiremos este mensaje cuando el dispositivo se esté quedando sin memoria. Cuando esto ocurra, deberemos liberar tanta memoria como podamos (objetos que puedan ser fácilmente recreados más tarde).

6. Ejercicios de propiedades y colecciones

6.1. Propiedades

Vamos a añadir propiedades a la clase `UAPelicula` creada en la sesión anterior.

- a) Crea propiedades para cada variable de instancia definida en la clase. Las variables de instancia pueden eliminarse, o renombrarse para no confundirse con la propiedad.
- b) En `UAMasterViewController`, haz que la película sea una propiedad de la clase, en lugar de una variable de instancia.
- c) Haz que en cada celda, en lugar de mostrar la descripción de la película muestre sólo el título, accediendo a él a través de la propiedad definida anteriormente.

6.2. Listas

Vamos ahora a utilizar colecciones de datos en nuestra aplicación. Se pide:

- a) Añade a las películas una lista de actores. Esta lista debe poder ser accedida como propiedad.
- b) Ahora en lugar de guardar una única película vamos a tener una lista de ellas. Para ello en la clase `UAMasterViewController` sustituiremos la propiedad de tipo `UAPelicula` por una de tipo `NSArray`, y en el constructor crearemos una serie de películas de prueba y las añadiremos a la lista.
- c) Modifica el método `tableView:numberOfRowsInSection:`, y haz que devuelva el número de componentes de la lista, en lugar de un valor fijo.
- d) Modifica el método `tableView:cellForRowIndexPath:`, para que muestre como texto de la celda el título de la película correspondiente a la fila actual. Esta vez, accede a la propiedad `row` de `NSIndexPath` mediante el operador `'.'`. Comprueba que la lista de películas se muestra de forma correcta en la lista.

6.3. Temporizadores (*)

Crearemos ahora un temporizador con `NSTimer`, que transcurridos unos segundos escribirá un log.

- a) Cuando se lance la aplicación, programa un temporizador que se dispare en 10 segundos. Escribe un log en el momento en el que se programa, y otro en el momento en el que se dispara (dentro del método programado).
- b) Haz ahora que al dispararse el temporizador se envíe una notificación con identificador

"TemporizadorCompletado".

c) En la clase `UAMasterViewController` registra un método como observador de la notificación "TemporizadorCompletado". En dicho método haz que se añada una nueva película a la lista, y actualiza la interfaz llamando a `[self.tableView reloadData]`.

d) Modifica el observador anterior para utilizar un bloque, en lugar de un método.

e) Haz que el observador anterior añada a la lista de películas un objeto de tipo `NSString`, en lugar de `UAPelicula`. Al mostrar la lista de películas, cuando se añada la cadena dará un error, ya que estamos intentando mostrar la propiedad `item.titulo`, que no existe en `NSString`. Modificar el código de `tableView:cellForRowAtIndexPath:` para que compruebe si el item de la lista responde al selector `titulo`. De no ser así, mostraremos como texto la descripción del propio objeto.

6.4. Gestión de memoria con ARC

Vamos a convertir el proyecto anterior a ARC para que la gestión de la memoria la haga automáticamente el compilador.

a) Realizar un *snapshot* del proyecto antes de realizar la conversión.

b) Convertir el proyecto de forma automática con la opción *Edit > Refactor > Convert to Objective-C ARC* ¿Qué cambios se producen?

7. Vistas

7.1. Patrón Modelo-Vista-Controlador (MVC)

La API de Cocoa está claramente basada en este patrón, por lo que resulta apropiado hacer que nuestras aplicaciones también lo adopten. Este patrón se basa en dividir nuestras funcionalidades en tres grandes bloques: el modelo, la vista, y el controlador.

- **Modelo:** Aquí reside toda la lógica de negocio y de acceso a datos de nuestra aplicación. En esta parte encontramos los objetos del dominio de nuestra aplicación, como por ejemplo `UAAsignatura`, y todos aquellos que nos permitan acceder y manipular estos datos.
- **Vista:** En la vista tenemos los componentes que se muestran en la interfaz. Estos componentes serán las clases del *framework UIKit* cuyo nombre tiene el sufijo `View`, y que derivan de la clase `UIView`. Normalmente crearemos estos objetos de forma visual mediante la herramienta Interface Builder incluida en Xcode, aunque también se podrán crear de forma programática. A parte de los tipos de vistas que podemos encontrar en la API de Cocoa Touch, podremos crear vistas propias definiendo nuevas subclases de `UIView`.
- **Controlador:** El controlador es el que se encarga de coordinar modelo y vista. Aquí es donde definimos realmente el comportamiento de nuestra aplicación. Si bien modelo y vista deben diseñarse de forma que sean lo más reutilizables posible, el controlador está fuertemente vinculado a nuestra aplicación concreta.

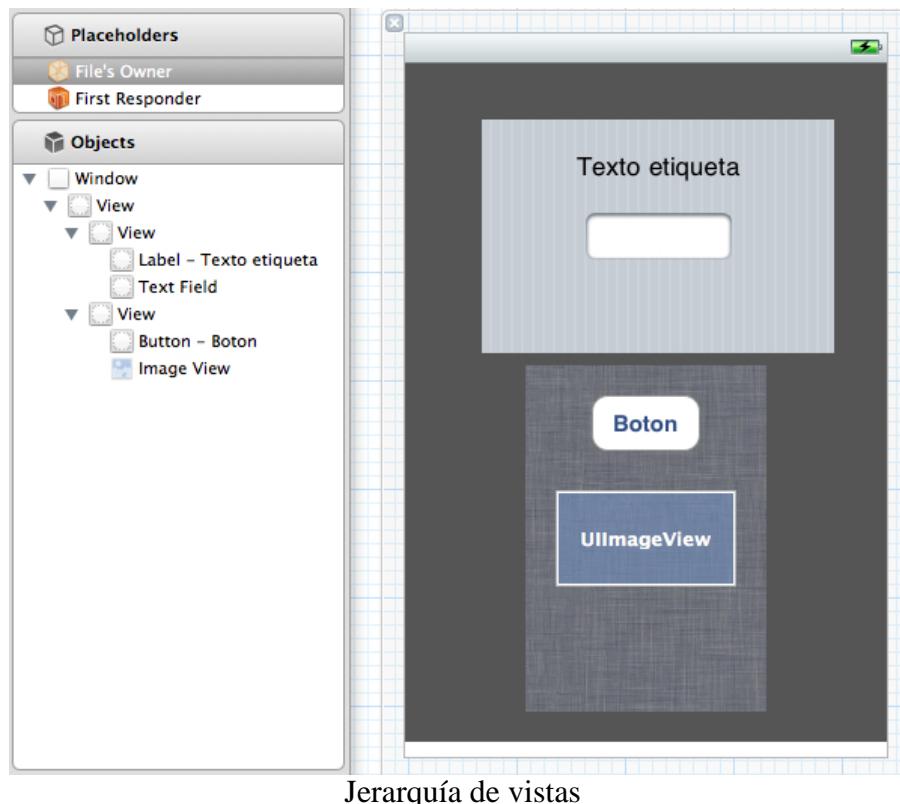
Por el momento hemos comenzado viendo cómo implementar las clases del dominio de nuestra aplicación, que formarán parte del modelo. Ahora vamos a centrarnos en la creación de la vista.

7.2. Jerarquía de vistas, ventanas, y controladores

Como hemos comentado, las clases de la vista derivan todas de `UIView`, y en ellas se especifica cómo debe mostrarse dicha vista en la pantalla. Una vista define una región rectangular (*frame*) que se mostrará en la pantalla del dispositivo, y en el código de su clase se indica cómo debe dibujarse el contenido en dicha región. Además de encargarse de mostrar el contenido, la vista se encarga también de interactuar con el usuario, es lo que se conoce como un *responder*. La clase `UIView` deriva de `UIResponder`, es decir, la vista también se encarga de responder ante los eventos (contactos en la pantalla táctil) que el usuario realice en su región. Podemos encontrar una serie de vistas ya predefinidas en la API de Cocoa Touch que podremos utilizar en nuestras aplicaciones (bien añadiéndolas mediante *Interface Builder* o bien añadiéndolas de forma programática), como son por ejemplo `UIImageView`, `UITextField`, `UILabel`, `UIButton` o `UIWebView`, entre otras muchas. Se suele tener como convención poner a las vistas el sufijo `View`, aunque vemos que no siempre es así. También podremos crearnos nuestra propia subclase de `UIView`.

para definir una vista personalizada, en la que nosotros decidamos cómo se dibuja el contenido de su región y cómo responde ante los eventos del usuario.

Las vistas se organizan de forma jerárquica. Una vista puede contener subvistas, de forma que cualquier modificación que se haga sobre una vista afectará también a todas sus subvistas. En la raíz de la jerarquía tendremos un tipo especial de vista conocida como ventana, que se define en la clase `UIWindow`. Normalmente nuestra aplicación tendrá sólo una ventana, que ocupará toda la pantalla (podremos tener más de una ventana si por ejemplo nuestra aplicación soporta tener una pantalla externa). En la aplicación veremos aquellas vistas que sean subvistas de la ventana principal.



Jerarquía de vistas

Nota

Una subvista no está restringida a la región que ocupa su vista padre, sino que puede mostrarse fuera de ella. Por lo tanto, no se puede determinar la jerarquía de vistas a partir del contenido que veamos en la pantalla. Sin embargo, si que afecta al orden en el que se dibujan (una vista se dibuja antes que sus subvistas, por lo que las subvistas taparán el contenido de la vista padre).

Normalmente en la ventana principal tendremos como hija una vista que abarcará toda la ventana, y dentro de ella encontraremos los diferentes objetos de la interfaz. Habitualmente no tendremos directamente un objeto de tipo `UIView`, sino que lo que encontraremos dentro de la ventana es un controlador. Los controladores se encargan de

gestionar el ciclo de vida de las vistas y los cambios que se puedan producir en ellas (como por ejemplo ajustarlas al cambiar la orientación del dispositivo). No deberemos crear un controlador por cada vista individual que se muestre en la pantalla, sino por cada pantalla de la aplicación (que se compondrá de una jerarquía de vistas). Lo que podemos encontrar habitualmente es un controlador que hace de contenedor de otros controladores (por ejemplo las aplicaciones basadas en navegación o en pestañas se componen de varias pantallas, tendremos un controlador general que se encarga de gestionar la pila de navegación o las pestañas, y a parte un controlador específico para cada pantalla).

Los controladores derivan de la clase `UIViewController`, y tienen una propiedad `view` que corresponde a la vista que mostrarán. Cuando añadamos un controlador a la ventana, se mostrará en ella la vista referenciada por dicha propiedad. Modificando su valor cambiaremos la vista que muestra el controlador.

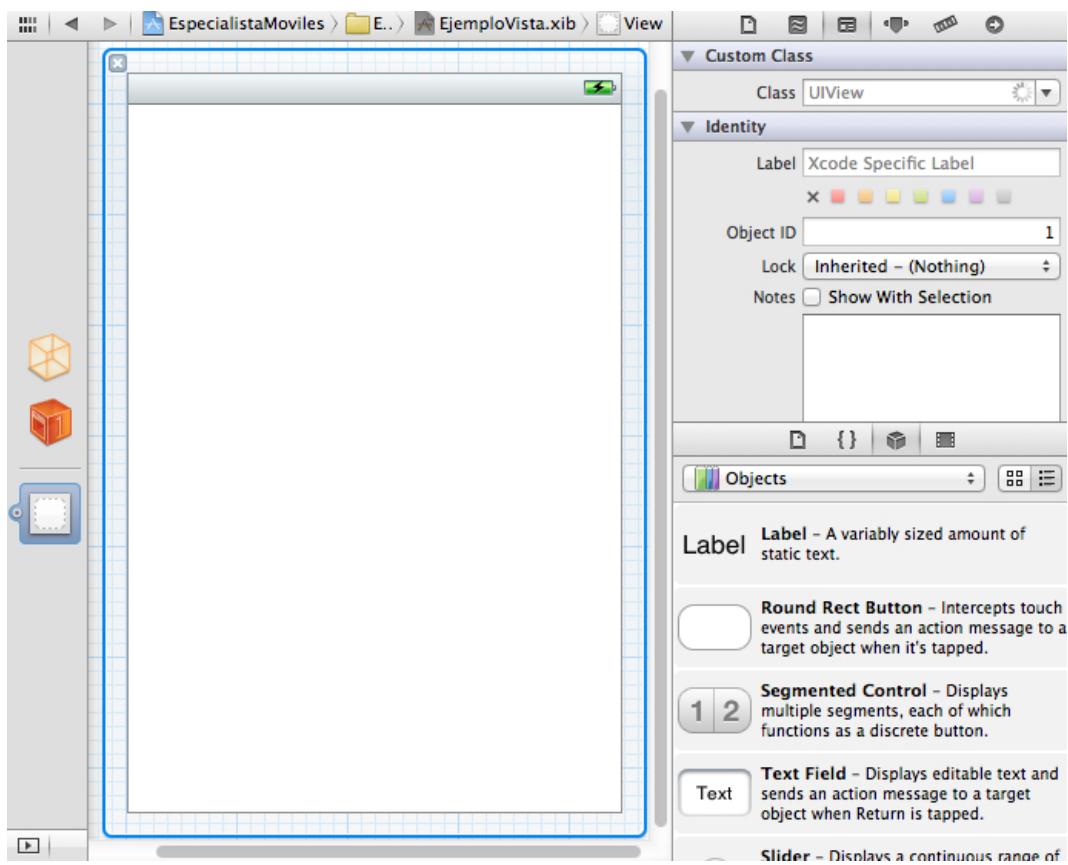
Si hemos utilizado la plantilla `>Master-Detail Application` a la hora de crear nuestro proyecto, veremos que nos ha creado dos controladores para la pantalla principal: *Navigation controller*, que se crea de forma programática en nuestra `UIApplicationDelegate`, está implementado en la API de Cocoa Touch y es el que nos permite tener aplicaciones con una barra de navegación y una pila de pantallas; y *MasterViewController* que corresponde a la pantalla principal en la navegación y se implementa dentro de nuestra aplicación. También tenemos un tercer controlador (*DetailViewController*) que corresponde a la pantalla secundaria de la navegación en la que entraremos cuando se pulse un ítem de la pantalla principal.



7.3. Creación de vistas con Interface Builder

Lo más habitual será crear la interfaz de nuestra aplicación de forma visual con Interface Builder. En versiones anteriores de Xcode, Interface Builder se proporcionaba como una herramienta independiente, pero a partir de Xcode 4 está totalmente integrada en el entorno. Los ficheros en los que se define la interfaz se denominan ficheros NIB (o XIB). En las plantillas que hemos creado anteriormente los hemos visto como fichero .xib, aunque si construimos la aplicación y miramos el contenido del *bundle* generado, veremos que se empaqueten como .nib, ya que esta última es la versión compilada del fichero .xib anterior.

El fichero NIB simplemente es un contenedor de objetos de la interfaz, que podrá cargarse en el código de nuestra aplicación y así tener disponibles dichos objetos para mostrar. Si pulsamos sobre un fichero .xib en el navegador de Xcode, veremos que en el editor se abre Interface Builder. Vamos a estudiar los elementos que encontramos en dicha herramienta.



Aspecto de Interface Builder

- *Dock de objetos:* A la izquierda del editor veremos los iconos de los objetos que se

definen (o se referencian) en el fichero NIB. Seleccionando uno de ellos podremos visualizarlo y modificar sus propiedades.

- *Editor visual*: En la parte central se presentan de forma visual los componentes definidos en el fichero NIB.
- *Inspector y librería*: A la izquierda tenemos el inspector, donde podemos ver y modificar las propiedades del objeto seleccionado en un momento dado. Bajo el inspector tenemos la librería de objetos que podemos añadir al NIB, simplemente arrastrándolos sobre el *dock* o sobre el editor.

Podemos crear un nuevo NIB en nuestro proyecto seleccionando *File > New > New File ... > iOS > User Interface* y dentro de esta sección seleccionaremos una de las plantillas disponibles. Podemos crear un NIB vacío, o bien crearlo con alguno de los elementos básicos ya añadidos (una vista, una ventana, o un *Application Delegate*). Lo habitual será crear un NIB junto a un controlador, como veremos en la próxima sesión, pero también puede interesarnos crear únicamente el fichero con los objetos de la interfaz.

7.3.1. Dock de objetos

Dentro del *dock* encontramos dos tipos de objetos, separados por una línea horizontal:

- **Objetos referenciados por el NIB**: Son objetos que ya existen en nuestra aplicación sin que tengan que ser creados por el NIB, simplemente se incluyen como referencia para establecer comunicaciones con ellos, y así poder comunicar la vista con el código de nuestra aplicación.
- **Objetos instanciados por el NIB**: Estos son los objetos que se crearán cuando carguemos el NIB desde nuestro código. Aquí es donde encontramos los componentes de la interfaz que se crearán.

Pinchando sobre cualquier objeto del *dock* podremos ver sus propiedades en el panel de la izquierda. De los objetos referenciados que encontramos en el *dock* podemos destacar:

- *File's Owner*: Referencia el objeto propietario de los objetos del NIB. Cuando se carga un NIB, siempre hay que especificar un objeto como propietario de los contenidos del NIB. El propietario debe existir antes de que se cargue el NIB.
- *First responder*: El *first responder* es el objeto que responderá en primera instancia a los eventos del usuario, si no se ha especificado otra forma de tratar los eventos. Realmente el elemento *First responder* del NIB es un objeto ficticio que nos permite indicar que los eventos los trate el *first responder* por defecto que tengamos en cada caso. Más adelante veremos esto con mayor detallamiento.

Respecto a los objetos que son instanciados por el NIB, podemos distinguir los siguientes:

- **Vistas y ventanas**: Objetos a mostrar en la interfaz, todos ellos subtipos de `UIView`. Se organizan de forma jerárquica. En el *dock* veremos los objetos en la cima de la jerarquía. Si desplegamos el *dock* podremos ver la jerarquía completa de objetos.
- **Objetos delegados**: Objetos pertenecientes a clases definidas por nosotros en los que

delegamos para tratar los eventos de los componentes de la interfaz. Definiendo aquí estos objetos, conseguiremos que se instancien de forma automática al cargar el NIB.

- **Controladores:** Objetos que se encargan de gestionar el comportamiento de las vistas. Podemos utilizar controladores ya implementados en Cocoa Touch para implementar estilos de navegación ya predefinidos, o bien controladores propios creados como una subclase de `UIViewController` dentro de nuestra aplicación.

Nota

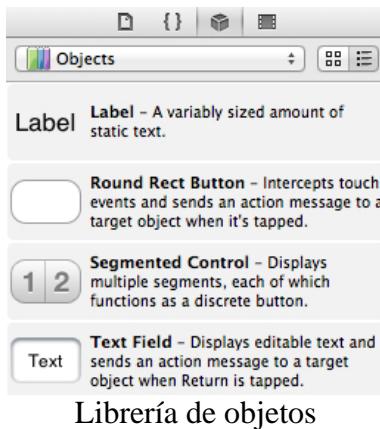
Podemos expandir o contraer el *dock* con el botón en forma de triángulo que encontramos en su esquina inferior izquierda. En la vista normal veremos sólo los objetos de nivel superior, pero en la vista expandida podremos desplegar toda la jerarquía de objetos.

7.3.2. Inspector y librería

Cuando tengamos abierto en el editor un fichero NIB en el panel de utilidades (lateral derecho) veremos algunas pestañas adicionales que no estaban mientras editábamos código fuente. En la parte superior (inspector), a parte de *File Inspector* (nombre del fichero `.xib`, tipo, ruta, localización, pertenencia a *targets*, etc) y *Quick Help inspector* (ayuda rápida sobre los objetos seleccionados en el *dock* o en el editor), tenemos tres inspectores adicionales, que nos van a dar información sobre el objeto que tengamos seleccionado en un momento dado:

- *Identity inspector*: Identidad del objeto. Nos permite indicar su clase (atributo *Class*). También podemos asignarle una etiqueta para identificarlo dentro de Xcode.
- *Attributes inspector*: Atributos del objeto seleccionado. Aquí es donde podemos editar los atributos de la vista seleccionada (color de fondo, fuente, estilo, etc).
- *Size inspector*: Información sobre el tamaño y la posición del objeto en pantalla. Nos permite definir estos atributos de forma que se ajusten de forma automática según la disposición de la pantalla.
- *Connections inspector*: Nos permite definir conexiones entre los diferentes objetos del *dock*. Por ejemplo, podremos vincular una vista definida en el NIB con un campo del *File's Owner*, de forma que desde el código de la aplicación podremos tener acceso a dicha vista.

Bajo el inspector encontramos la librería de objetos. Esta librería se puede utilizar también para añadir código, pero cobra mayor importancia cuando trabajemos con el editor visual. De las distintas pestañas de la librería, nos interesará *Object library*. En ella encontraremos los diferentes tipos de objetos que podremos añadir a nuestro NIB (vistas, ventanas, controles, controladores, objetos y objetos externos). Podemos añadir cualquiera de estos objetos simplemente arrastrándolo al *dock* o al editor.



7.3.3. Objeto propietario del NIB

Si creamos una vista en el NIB, nos puede interesar tener acceso a ella desde nuestro código para mostrarla en pantalla o modificar su contenido. El lugar desde donde accederemos a dichas propiedades será habitualmente el objeto que definamos como *File's Owner*, que será un objeto que ya existe en nuestra aplicación (puede ser de cualquier tipo).

Al cargar un NIB, especificaremos como parámetros, además del nombre del NIB a cargar, el objeto que se va a comportar como propietario:

```
[[NSBundle mainBundle] loadNibNamed: @"NombreNib"
                                owner: self
                               options: nil];
```

Nota

Esta no es la forma más habitual de cargar un fichero NIB. Normalmente crearemos un controlador asociado al fichero NIB, y el NIB será cargado automáticamente desde el controlador, como veremos en la siguiente sesión. Sin embargo, por ahora vamos a cargar manualmente el fichero NIB para así entender mejor la forma en la que se realiza la carga.

Podemos observar que estamos cargando el NIB entre los recursos empaquetados con la aplicación (*main bundle*). Para tener acceso a dicho paquete de recursos incluidos con la aplicación tenemos el *singleton* `[NSBundle mainBundle]`.

En el ejemplo hemos puesto como objeto propietario el objeto desde el que estamos realizando la llamada (`self`), pero podríamos especificar cualquier otro. Lo que vemos es que la carga del fichero NIB no nos está devolviendo nada, entonces, ¿cómo podemos acceder a los objetos cargados? Para acceder a ellos necesitaremos vincular dichos objetos con propiedades de la clase propietaria, para que así tras cargarlo podamos acceder a ellos a través de esas propiedades. Esta vinculación se establecerá a través de lo que se conocen como *outlets*.

7.3.4. Conexiones con outlets

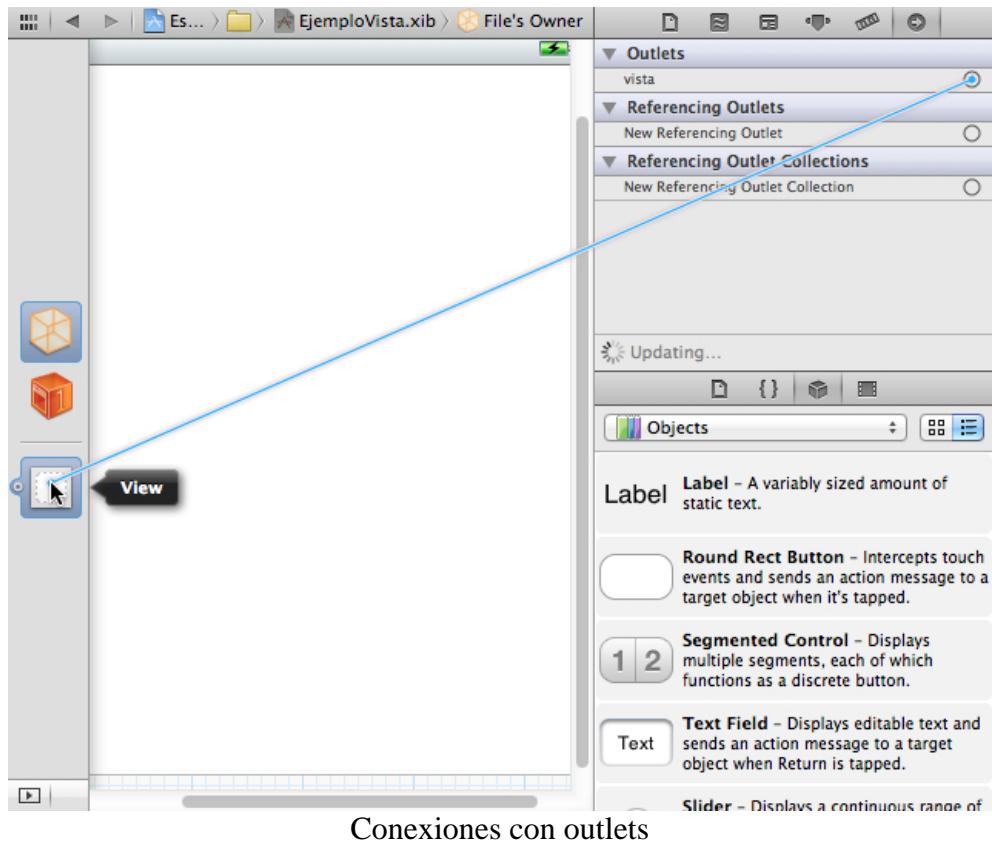
Los *outlets* nos permiten crear conexiones entre distintos objetos definidos o referenciados dentro de un fichero NIB. Podemos por ejemplo vincular distintos objetos instanciados en el NIB con las propiedades del *File's Owner*.

En primer lugar, deberemos asegurarnos de que en el NIB la clase del *File's Owner* está configurada correctamente. Esto lo tendremos en el atributo *Class* del *Identity inspector*. Por ejemplo supongamos que tenemos una clase a la que llamamos `EjemploVista` que será la que asignemos como propietaria al cargar el NIB. En ese caso, en el NIB tendremos que indicar que la identidad del *File's Owner* es de clase `EjemploVista`.

Ahora podemos declarar una serie de *outlets* en dicha clase. Los *outlets* son propiedades de la clase que *Interface Builder* reconocerá y que nos permitirá vincular con los diferentes objetos del NIB. Por ejemplo, si queremos vincular un objeto de tipo `UIView`, deberemos declarar en nuestra clase `EjemploVista` una propiedad de dicho tipo, y añadir el tipo `IBOutlet` a la declaración:

```
@interface EjemploVista : NSObject
@property(nonatomic,retain) IBOutlet UIView *vista;
@end
```

El tipo `IBOutlet` no tiene ningún efecto sobre la compilación, de hecho es eliminado en tiempo de preprocesamiento. Simplemente servirá para indicar a *Interface Builder* que dicha propiedad es un *outlet*. De esta forma, si ahora al editar el NIB seleccionamos *File's Owner* y *Connections inspector*, veremos que nos aparece en la lista el *outlet* `vista`. Podemos pinchar en el círculo junto al *outlet* y arrastrar hacia el objeto con el que queramos vincularlo. Deberemos conectarlo con un objeto de tipo `UIView`, ya que es así como lo hemos declarado.



Conexiones con outlets

Una vez hemos establecido dicha conexión, cuando carguemos el NIB como hemos visto anteriormente, especificando como propietario un objeto de clase `EjemploVista`, la vista (`UIView`) creada por el NIB será asignada a la propiedad `vista` de nuestro objeto `EjemploVista`.

Una consideración importante a la hora de cargar el NIB es cómo se realiza la gestión de la memoria de los objetos instanciados en dicho fichero. Siguiendo la regla de liberar quien reserva, si el NIB instancia, el NIB deberá liberar, por lo que tras cargar el NIB habremos obtenido los objetos de nivel superior con un *autorelease* pendiente que los borrará de la memoria si nadie los retiene (de igual forma que cuando utilizamos un método factoría). Por lo tanto, será responsabilidad del fichero propietario retener dichos objetos si quiere poder utilizarlos más adelante. Esto lo haremos declarando las propiedades asociadas a estos objetos con el modificador `strong`.

Deberemos retener todos los objetos de nivel superior. Sin embargo, los objetos de niveles inferiores de la jerarquía serán retenidos por sus padres, por lo que no será necesario retenerlos en el objeto propietario. Con retener los objetos de nivel superior será suficiente.

Para evitar tener referencias cíclicas deberemos seguir la siguiente regla: las referencias a los *outlets* serán fuertes sólo en el *File's Owner*, en cualquier otro objeto estas referencias

siempre deberán ser débiles.

Nota

Hemos visto cómo establecer conexiones entre propiedades de *File's Owner* y objetos definidos en el NIB. Sin embargo, los *outlets* se pueden aplicar a cualquier objeto definido o referenciado en el NIB.

7.3.5. Tratamiento de eventos mediante acciones

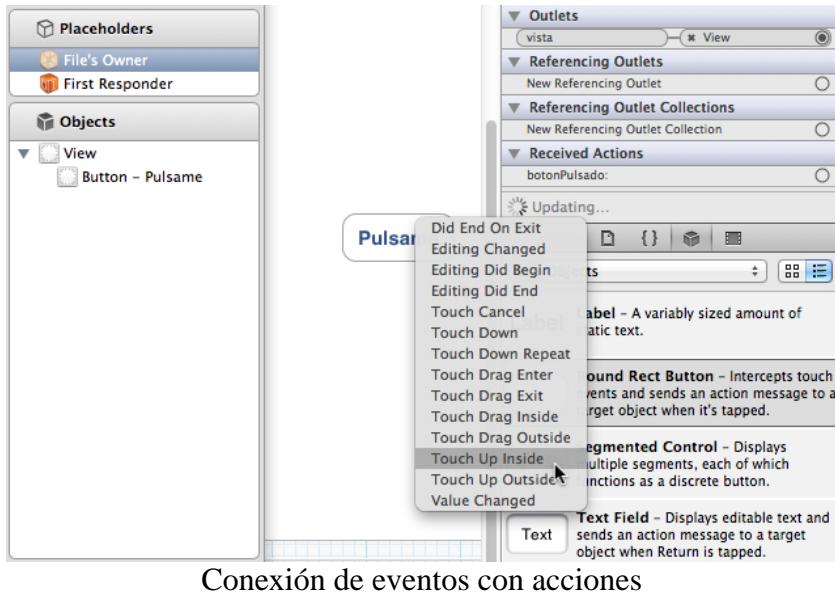
Con los *outlets* podemos vincular campos a objetos de la interfaz. Sin embargo, puede que nos interese que al realizar alguna acción sobre uno de dichos objetos de la interfaz se ejecute un método de nuestro objeto para darle respuesta al evento. Para hacer esto podemos utilizar las acciones. Los métodos declarados con el tipo `IBAction` podrán ser conectados con diferentes acciones de los objetos de la interfaz desde *Interface Builder*. Por ejemplo, podemos declarar un método de este tipo de la siguiente forma:

```
- (IBAction)botonPulsado: (id)sender;
```

Normalmente estos métodos recibirán un parámetro de tipo `id` en el que recibirán una referencia al objeto que generó el evento (por ejemplo, en el caso de conectar con el evento de pulsación de un botón, el objeto `UIButton` correspondiente al botón que fue pulsado). El tipo `IBAction` es sustituido en tiempo de compilación por `void`, pero servirá para que *Interface Builder* lo reconozca como una acción.

Vamos a suponer que añadimos el método anterior a la clase `EjemploVista` definida previamente. Si ahora añadimos dentro de la vista del fichero NIB un botón (`UIButton`), arrastrando un objeto de este tipo desde la librería de objetos, podremos conectar la acción de pulsar dicho botón con el método anterior. Esto podremos hacerlo de dos formas alternativas:

- Seleccionando *File's Owner*. En su inspector de conexiones veremos la acción `botonPulsado:` que podemos arrastrar sobre el botón para conectarla con él. Al soltar, nos aparecerá una lista de los posibles eventos que genera el botón. Cogeremos *Touch Up Inside*, que es el que se ejecuta cuando soltamos el botón estando dentro de él.
- Seleccionando el botón. En su inspector de conexiones vemos todos los eventos que puede enviar. Arrastramos desde el evento *Touch Up Inside* hasta *File's Owner* en el *dock*. Al soltar, nos saldrá una lista de las acciones definidas en dicho fichero (métodos que hemos declarado como `IBAction`). Seleccionaremos `botonPulsado:`



Conexión de eventos con acciones

Alternativa

También podemos establecer conexiones con acciones y *outlets* pulsando sobre el botón derecho (o *Ctrl-Click*) sobre los objetos del *dock* o del editor. La forma más rápida de crear estas conexiones consiste en utilizar la vista de asistente, para ver al mismo tiempo el fichero *.xib* y el fichero *.h* de su *File's Owner*. Si hacemos *Ctrl-Click* sobre la vista (en el *dock* o en el editor visual) y arrastramos sobre el código del fichero *.h*, nos permitirá crear automáticamente la declaración del método o de la propiedad correspondiente a la acción o al *outlet* respectivamente, y en el *.m* sintetizará la propiedad del *outlet* o creará el esqueleto de la función de la acción de forma automática.

7.3.6. Fichero de interfaz principal

Hemos visto que en las propiedades principales del proyecto (sección *Summary*) tenemos la opción de especificar un fichero *.xib* principal que se cargará cuando arranque la aplicación (*Main Interface*). En la plantilla creada esta opción no se utiliza, ya que la vista se está cargando de forma programática:

- En *main.m* a la función *UIApplicationMain* se le pasa como cuarto parámetro el nombre de la clase que hará de delegada de la aplicación. Al cargar la aplicación utilizará esta clase como delegada, y llamará a sus métodos cuando ocurran eventos del ciclo de vida de la aplicación
- En la clase delegada de la aplicación generada, en el evento *application:didFinishLaunchingWithOptions:* se crea la interfaz de forma programática y se muestra en pantalla.

Sin embargo, podríamos utilizar una forma alternativa de arrancar la aplicación:

- Creamos un fichero *.xib* utilizando la plantilla *Application*. Este plantilla contiene

todos los elementos necesarios para poder arrancar la aplicación a partir de dicho fichero de interfaz.

- Especificamos el fichero `.xib` creado en la propiedad *Main Interface* del proyecto.
- En `main.m`, dejamos el cuarto parámetro de `UIApplicationMain` a `nil`, para que no arranque directamente a partir del delegado de la aplicación, sino que coja la interfaz principal que hemos configurado.

El fichero de la interfaz principal que hemos creado con la plantilla *Application* contiene los siguientes elementos:

- El *File's Owner* es de tipo `UIApplication`, en este caso no es una clase de nuestra aplicación, sino que es una clase de la API de Cocoa Touch.
- Uno de los objetos instanciados por el fichero de interfaz es el tipo *App Delegate*. Deberemos hacer que este objeto sea del tipo concreto de nuestro objeto delegado de la aplicación.
- La clase `UIApplication` tiene un *outlet* llamado `delegate` que podremos vincular con el objeto *App Delegate*.
- Se instancia también la ventana principal de la aplicación. Esta ventana se conecta con un *outlet* en el *App Delegate*.

Podemos ver que los *outlets* que referencian los objetos de la interfaz en este caso no estarán definidos en el *File's Owner*, sino en el *App Delegate*, que además se trata de un objeto creado por el NIB (no referenciado). De esta forma, cuando la aplicación arranque `UIApplicationMain` cargará el contenido del NIB especificado, que instanciará al delegado de la aplicación y lo guardará en la propiedad `delegate` de `UIApplication`. A su vez, instanciará también la ventana principal que será guardada en un *outlet* del objeto delegado de la aplicación. El delegado deberá mostrar esta ventana en pantalla al iniciarse la aplicación.

7.4. Creación de vistas de forma programática

Hemos visto como crear la interfaz con Interface Builder, pero todo lo que se puede hacer con dicha herramienta se puede hacer también de forma programática, ya que lo único que hace Interface Builder es crear objetos de la API de Cocoa Touch (o definidos por nosotros) y establecer algunas de sus propiedades.

7.4.1. Ventanas

En primer lugar, podemos ver cómo podríamos crear la ventana principal desde nuestro código, instanciando un objeto de tipo `UIWindow` e indicando el tamaño del marco de la ventana:

```
UIWindow *window = [[UIWindow alloc]
    initWithFrame: [[UIScreen mainScreen] bounds]];
```

Lo habitual será inicializar la ventana principal con un marco del tamaño de la pantalla,

tal como hemos hecho en el ejemplo anterior.

Para mostrar la ventana en pantalla debemos llamar a su método `makeKeyAndVisible` (esto se hará normalmente en el método `application:didFinishLaunchingWithOptions:` del delegado de `UIApplication`). Esto se hará tanto cuando la ventana ha sido cargada de un NIB como cuando la ventana se ha creado de forma programática.

```
[self.window makeKeyAndVisible];
```

Es posible que necesitemos acceder a esta ventana, por ejemplo para cambiar la vista que se muestra en ella. Hacer esto desde el delegado de `UIApplication` es sencillo, porque normalmente contamos con la propiedad `window` que hace referencia a la pantalla principal, pero acceder a ella desde otros lugares del código podría complicarse. Para facilitar el acceso a dicha ventana principal, podemos obtenerla a través del *singleton* `UIApplication`:

```
UIWindow *window = [[UIApplication sharedApplication] keyWindow];
```

Si podemos acceder a una vista que se esté mostrando actualmente dentro de la ventana principal, también podemos obtener dicha ventana directamente a través de la vista:

```
UIWindow *window = [vista window];
```

7.4.2. Vistas

También podemos construir una vista creando un objeto de tipo `UIView`. En el inicializador deberemos proporcionar el marco que ocupará dicha vista. El marco se define mediante el tipo `CGRect` (se trata de una estructura, no de un objeto). Podemos crear un nuevo rectángulo con la macro `CGRectMake`. Por ejemplo, podríamos inicializar una vista de la siguiente forma:

```
UIView *vista = [[UIView alloc]
                  initWithFrame:CGRectMake(0,0,100,100)];
```

Como alternativa, podríamos obtener el marco a partir del tamaño de la pantalla o de la aplicación:

```
// Marco sin contar la barra de estado
CGRect marcoVista = [[UIScreen mainScreen] applicationFrame];

// Límites de la pantalla, con barra de estado incluida
CGRect marcoVentana = [[UIScreen mainScreen] bounds]
```

Normalmente utilizaremos el primero para las vistas que queramos que ocupen todo el espacio disponible en la pantalla, y el segundo para definir la ventana principal. Hemos de destacar que `UIWindow` es una subclase de `UIView`, por lo que todas las operaciones disponibles para las vistas están disponibles también para las ventanas. Las ventanas no son más que un tipo especial de vista.

Las vistas (`UIView`) también nos proporcionan una serie de métodos para consultar y

modificar la jerarquía. El método básico que necesitaremos es `addSubview`, que nos permitirá añadir una subvista a una vista determinada (o a la ventana principal):

```
[self.window addSubview: vista];
```

Con esto haremos que la vista aparezca en la pantalla. Podemos también eliminar una vista enviándole el mensaje `removeFromSuperview` (se le envía a la vista hija que queremos eliminar). Podemos también consultar la jerarquía con los siguientes métodos:

- `superview`: Nos da la vista padre de la vista destinataria del mensaje.
- `subviews`: Nos da la lista de subvistas de una vista dada.
- `isDescendantOfView`: Comprueba si una vista es descendiente de otra.

Como vemos, una vista tiene una lista de vistas hijas. Cada vista hija tiene un índice, que determinará el orden en el que se dibujan. El índice 0 es el más cercano al observador, y por lo tanto tapará a los índices superiores. Podemos insertar una vista en un índice determinado de la lista de subvistas con `insertSubviewAtIndex:`.

Puede que tengamos una jerarquía compleja y necesitemos acceder desde el código a una determinada vista por ejemplo para inicializar su valor. Una opción es hacer un *outlet* para cada vista que queramos modificar, pero esto podría sobrecargar nuestro objeto de *outlets*. También puede ser complejo y poco fiable el buscar la vista en la jerarquía. En estos casos, lo más sencillo es darle a las vistas que buscamos una etiqueta (*tag*) mediante la propiedad *Tag* del inspector de atributos, o asignando la propiedad `tag` de forma programática. Podremos localizar en nuestro código una vista a partir de su etiqueta mediante `viewWithTag`. Llamando a este método sobre una vista, buscará entre todas las subvistas aquella con la etiqueta indicada:

```
UIView *texto = [self.window viewWithTag: 1];
```

Podemos también obtener la posición y dimensiones de una vista. Tenemos varias opciones:

```
// Límites en coordenadas locales
// Su origen siempre es (0,0)
CGRect area = [vista bounds];

// Posición del centro de la vista en coordenadas de su supervista
CGPoint centro = [vista center];

// Marco en coordenadas de la supervista
CGRect marco = [vista frame]
```

A partir de `bounds` y `center` podremos obtener `frame`.

7.4.3. Acciones y eventos

Cuando creamos vistas de forma programática ya no hace falta definir *outlets* para ellas, ya que la función de los *outlets* era permitirnos acceder a los contenidos del NIB desde código. Con las acciones pasará algo similar, ya no es necesario declarar los métodos como `IBAction`, ya que dicho tipo sólo servía para que Interface Builder lo reconociese

como una acción, pero si que tendremos que indicar a las vistas en código el *target* y el *selector* al que queremos que llame cuando suceda un evento.

Aunque no declaremos los *callbacks* con `IBAction`, deberemos hacer que devuelvan `void` (equivalente a `IBAction`), y que tomen como parámetro el objeto que generó el evento:

```
- (void)botonPulsado: (id) sender {  
    ...  
}
```

Esta es la forma más habitual del *callback*, aunque en su forma completa incluye también un segundo parámetro de tipo `UIEvent` con los datos del evento que se ha producido.

```
- (void)botonPulsado: (id) sender event: (UIEvent *) evento {  
    ...  
}
```

Ahora debemos conectar este método de forma programática con el objeto que genera los eventos. Esto lo haremos con `addTarget:action:forControlEvents:` de la clase `UIControl`, que es una subclase de `UIView` de la que derivan todos los controles (botones, campos de texto, *sliders*, etc). Como podemos ver, se utiliza el patrón *target-acción*.

```
[boton addTarget: self action: @selector(botonPulsado:)  
forControlEvents: UIControlEventTouchUpInside];
```

Podemos programar eventos poniendo como *target* `nil`:

```
[boton addTarget: nil action: @selector(botonPulsado:)  
forControlEvents: UIControlEventTouchUpInside];
```

En este caso, estamos indicando que cuando se produzca el evento se llame al selector indicado en el objeto que se comporte como *first responder* del control. Un *responder* es un objeto que deriva de la clase `UIResponder`, y que por lo tanto sabe como responder ante eventos de la interfaz. La clase `UIView` hereda de `UIResponder`, por lo que cualquier vista puede actuar como *responder*.

Cuando tenemos una jerarquía de vistas se forma una cadena de *responders*. La cadena comienza desde la vista en la que se produce el evento, tras ello sube a su vista padre, y así sucesivamente hasta llegar a la ventana principal y finalmente a la aplicación (`UIApplication`), que también es un *responder*.

Cuando definamos una acción cuyo *target* es `nil`, se buscará el *selector* indicado en la cadena de *responders* en el orden indicado. Si en ninguno de ellos se encuentra dicho *selector*, el evento quedará sin atender.

Este tipo de acciones dirigidas a `nil` también pueden ser definidas con *Interface Builder*. Para ello encontramos entre los objetos referenciados uno llamado *First Responder*. Realmente, dicha referencia es `nil`, de forma que cuando conectemos un evento a dicho objeto, realmente estaremos definiendo una acción a `nil`.

7.5. Propiedades de las vistas

A continuación vamos a repasar las propiedades básicas de las vistas, que podremos modificar tanto desde Interface Builder como de forma programática.

7.5.1. Disposición

Entre las propiedades más importantes en las vistas encontramos aquellas referentes a su disposición en pantalla. Hemos visto que tanto cuando creamos la vista con Interface Builder como cuando la inicializamos de forma programática hay que especificar el marco que ocupará la vista en la pantalla. Cuando se crea de forma visual el marco se puede definir pulsando con el ratón sobre los márgenes de la vista y arrastrando para así mover sus límites. En el código estos límites se especifican mediante el tipo `CGRect`, en el que se especifica posición `(x,y)` de inicio, y el ancho y el alto que ocupa la vista. Estos datos se especifican en el sistema de coordenadas de la supervista.

Sin embargo, en muchas ocasiones nos interesa que el tamaño no sea fijo sino que se adapte al área disponible. De esta forma nuestra interfaz podría adaptarse de forma sencilla a distintas orientaciones del dispositivo (horizontal o vertical).

Para que la disposición de los elementos de la pantalla se ajuste automáticamente al espacio disponible, deberemos activar la propiedad `autoresizedSubviews` de la supervista que contiene dichos elementos. Esta propiedad se puede encontrar también en el inspector de atributos de Interface Builder con el nombre *Autoresizing Subviews*.

Ahora necesitamos especificar la forma en la que las diferentes subvistas van a ajustar su tamaño. Para ello definimos aquellos ejes que pueden tener tamaño flexible y aquellos que son de tamaño fijo. Esto lo podemos ajustar de forma visual desde el *size inspector*. Tendremos ejes internos de la vista (horizontal y vertical), y ejes externos (superior, inferior, derecho e izquierdo).

Los ejes internos marcados en rojo serán flexibles, mientras que los que no lo estén serán de tamaño fijo. Para los ejes externos el comportamiento es el contrario, los ejes marcados en rojo son de tamaño fijo, mientras que los que no estén marcados serán de tamaño flexible.

Esto se puede hacer también desde el código, mediante la propiedad `autoresizingMask` de la vista.

7.5.2. Transformaciones

Podemos también aplicar una transformación a las vistas, mediante su propiedad `transform`. Por defecto las vistas tendrán aplicada la transformación identidad `CGAffineTransformIdentity`.

La transformación se define mediante una matriz de transformación 2D de dimensión 3x3. Podemos crear transformaciones de forma sencilla con macros como `CGAffineTransformMakeRotation` o `CGAffineTransformMakeScale`.

Atención

Si nuestra vista tiene aplicada una transformación diferente a la identidad, su propiedad `frame` no será significativa. En este caso sólo deberemos utilizar `center` y `bounds`.

7.5.3. Otras propiedades

En las vistas encontramos otras propiedades que nos permiten determinar su color o su opacidad. En primer lugar tenemos `backgroundColor`, con la que podemos dar el color de fondo de una vista. En el inspector de atributos (sección *View*) podemos verlo como propiedad *Background*. El color de fondo puede ser transparente, o puede utilizarse como fondo un determinado patrón basado en una imagen.

De forma programática, el color se especifica mediante un objeto de clase `UIColor`. En esta clase podemos crear un color personalizado a partir de sus componentes (rojo, verde, azul, alpha), o a partir de un patrón.

Por otro lado, también podemos hacer que una vista tenga un cierto grado de transparencia, o esté oculta. A diferencia de `backgroundColor`, que sólo afectaba al fondo de la vista, con la propiedad `alpha`, de tipo `CGFloat`, podemos controlar el nivel de transparencia de la vista completa con todo su contenido y sus subvistas. Si una vista no tiene transparencia, podemos poner su propiedad `opaque` a `YES` para así optimizar la forma de dibujarla. Esta propiedad sólo debe establecerse a `YES` si la vista llena todo su contenido y no deja ver nada del fondo. De no ser así, el resultado es impredecible. Debemos llevar cuidado con esto, ya que por defecto dicha propiedad es `YES`.

Por último, también podemos ocultar una vista con la propiedad `hidden`. Cuando hagamos que una vista se oculte, aunque seguirá ocupando su correspondiente espacio en pantalla, no será visible ni recibirá eventos.

7.6. Controles básicos

En Cocoa Touch podemos encontrar una serie de vistas y controles que podemos utilizar en nuestras aplicaciones. Todos estos elementos están disponibles en la librería de objetos, y podemos añadirlos a la interfaz simplemente arrastrándolos sobre el NIB. Vamos a ver algunos de ellos.

7.6.1. Etiquetas de texto

Una de las vistas más sencillas es la etiqueta de texto. Con ella podemos mostrar texto estático en la interfaz. En esta vista fundamentalmente configuraremos el texto a mostrar

y sus propiedades (las encontraremos en la sección *Label* del inspector de atributos). También podemos crear esta vista de forma programática creando una instancia de `UILabel`.

```
UILabel *etiqueta = [[UILabel alloc]
    initWithFrame: CGRectMake(0,0,200,30)];
```

Nota

Podemos ver que todas las vistas se crean de forma programática utilizando el inicializador designado de su superclase `UIView`, tal como hemos visto anteriormente.

Es posible que desde el código necesitemos establecer el texto a mostrar en la etiqueta. Para ello utilizaremos su propiedad `text`.

```
[etiqueta setText: @"Titulo"];
```

7.6.2. Campo de texto

Un campo de texto nos proporciona un espacio donde el usuario puede introducir y editar texto. Se define en la clase `UITextField`, y pertenece a un grupo de vistas denominados controles, junto a otros componentes como por ejemplo los botones. Esto es así porque permiten al usuario interactuar con la aplicación. No heredan directamente de `UIView`, sino de su subclase `UIControl`, que incorpora los métodos para tratar eventos de la interfaz mediante el patrón *target-acción* como hemos visto anteriormente.

Sus propiedades se pueden encontrar en la sección *Text Field* del inspector de atributos. Podremos especificar un texto por defecto (*Text*), o bien un texto a mostrar sombreado en caso de que el usuario no haya introducido nada (*Placeholder Text*). Esto será útil por ejemplo para dar una pista al usuario sobre lo que debe introducir en dicho campo.

Si nos fijamos en el inspector de conexiones del campos de texto, veremos la lista de eventos que podemos conectar a nuestra acciones. Esta lista de eventos es común para cualquier control. En el caso de un campo de texto por ejemplo nos puede interesar el evento *Value Changed*.

7.6.3. Botones

Al igual que los campos de texto, los botones son otro tipo de control (heredan de `UIControl`). Se definen en la clase `UIButton`, que puede ser inicializada de la misma forma que el resto de vistas.

Si nos fijamos en el inspector de atributos de un botón (en la sección *Button*), vemos que podemos elegir el tipo de botón (atributo *Type*). Podemos seleccionar una serie de estilos prefijados para los botones, o bien darle un estilo propio (*Custom*).

El texto que aparece en el botón se especifica en la propiedad *Title*, y podemos configurar

también su color, sombreado, o añadir una imagen como ícono.

En el inspector de conexiones, el evento que utilizaremos más comúnmente en los botones es *Touch Up Inside*, que se producirá cuando levantemos el dedo tras pulsar dentro del botón. Este será el momento en el que se realizará la acción asociada al botón.

7.6.4. Imágenes

Para mostrar imágenes en la interfaz de la aplicación lo más sencillo es utilizar la vista `UIImageView`, que se encarga de mostrar una imagen estática. Deberemos indicar la imagen en su atributo *Image*.

Si creamos esta vista de forma programática, deberemos especificar la imagen como un objeto de clase `UIImage`. La forma más sencilla de instanciar un objeto de este tipo es mediante su método factoría `imageNamed:`, en el que debemos proporcionar el nombre del fichero de la imagen, que buscará en el *bundle* principal de la aplicación (esto es equivalente a especificar la imagen en el inspector de atributos).

```
UIImage *imagen = [UIImage imageNamed: @"imagen.png"];
UIImageView *imageView = [[UIImageView alloc] initWithImage: imagen];
```

En este caso los límites de la vista se ajustarán al tamaño de la imagen.

Deberemos añadir las imágenes como recursos al proyecto, y dentro del *target* deberán aparecer dentro de la fase *Copy Bundle Resources*. Cuando añadamos un recurso de tipo imagen lo normal será que automáticamente sea añadido a dicha fase. La clase `UIImage` soporta numerosos formatos de imagen: TIFF, JPEG, GIF, PNG, DIB, ICO, CUR, XBM.

Pantalla retina

Si queremos que nuestra aplicación aproveche la resolución de la pantalla retina, sin malgastar memoria en dispositivos de resolución inferior, podemos proporcionar dos versiones de la imagen, una de ellas con sufijo `@2x`. Por ejemplo, podríamos tener `logo.png` y `logo@2x.png`. En el código (o en Interface Builder) especificaremos siempre como imagen `logo.png`, pero la API de Cocoa Touch cargará una versión u otra según si el dispositivo cuenta con pantalla retina o no.

8. Ejercicios de vistas

8.1. Creación de la vista con Interface Builder

Vamos a crear la pantalla con los detalles de una película utilizando Interface Builder. En la plantilla nos ha creado una vista para los detalles en el fichero `UADetailViewController.xib`. Abriremos dicho fichero y realizaremos los siguientes pasos:

a) Vamos a borrar el contenido actual de la vista (la etiqueta `UILabel` incluida por la plantilla). Configuraremos la vista principal para que simule tener una barra de navegación en la parte superior y una barra de pestañas en la parte inferior (esto puede hacerse desde el inspector de atributos). Con esto podremos diseñar la pantalla sabiendo el espacio con el que contaremos realmente.

b) Creamos una pantalla con:

- `UILabel` para el *Título*.
- `UITextField` para el título. Debe mostrar como texto *placeholder* `Título de la película`, que es lo que se indicará cuando el usuario no haya introducido nada en este campo. La introducción de texto por defecto pondrá en mayúsculas la primera letra de cada palabra, y el teclado tendrá la tecla *Done* para indicar que hemos terminado de introducir el texto. Todos estos elementos deben especificarse mediante el editor de atributos.
- `UILabel` para el *Director*.
- `UITextField` para el director. Tendrá los mismos atributos que el del título, y como texto de *placeholder* `Director de la película`.
- `UILabel` para *Calificación de edad*.
- `UISegmentedControl` para seleccionar la calificación de edad de la película. Tendrá 4 segmentos: TP, NR7, NR13 y NR18.
- `UILabel` para la *Valoración*
- `UISlide` para especificar la valoración.
- `UILabel` para indicar el valor especificado en la barra de valoración de forma numérica. Estará junto a dicha barra.
- `UIButton` para *Guardar* los datos y para *Ver cartel* de la película.

El aspecto de esta pantalla deberá ser el siguiente:



Vista de detalles de la película

c) Probar que la pantalla se ve correctamente al ejecutar la aplicación. Al editar el texto del título o del director, ¿puedes cerrar el teclado? Solucionaremos este problema en los próximos ejercicios.

8.2. Conexión de componentes mediante outlets

Vamos a conectar los elementos de la interfaz creados en el ejercicio anterior con nuestro código Objective-C. Se pide:

a) Crea una serie de *outlets* en `UADetailViewController` para:

- Los `UITextField` del título y el director.
- El `UISegmentedControl` de la calificación de edad.
- El `UISlider` de la puntuación y el `UILabel` que hay junto a él para indicar el valor numérico.

b) Conecta en Interface Builder las vistas anteriores con los *outlets* que acabamos de crear.

c) Haz que al pasar a la pantalla de detalles de la película se rellenen los campos con los valores almacenados para la película. Para ello bástate en el método `setDetailItem` de la

plantilla, y modificalo para que lo que se proporcione sea una película (puedes cambiar el nombre al método y a la propiedad). En `configureView` rellenaremos los elementos de la interfaz a partir de los datos de la película. En `UIMasterViewController` localiza el método `tableView:didSelectRowAtIndexPath:` y haz que se proporcione al objeto `UIDetailViewController` los datos de la película de la fila seleccionada antes de mostrar dicha pantalla.

8.3. Controles y acciones

Vamos a tratar los diferentes eventos de los controles de la pantalla mediante acciones. Se pide:

a) En los campos de texto hemos visto que el teclado no se cierra cuando terminamos de editar el texto (cuando pulsamos el botón *Done*), lo cual es un problema. Para cerrar el teclado asociado a un campo de texto debemos hacer que el campo de texto deje de ser *First responder*, mediante el siguiente método:

```
[campo resignFirstResponder];
```

Esto deberemos hacerlo cuando se termine de editar el texto. Esto corresponde al evento *Did end on exit* del `UITextField`. Añade la acción correspondiente en `UADetailViewController`, y conéctala con el correspondiente evento en los dos campos de texto (título y director). Comprueba que ahora el teclado se cierra correctamente.

Alternativa

También podemos hacer esto creando un delegado de `UITextField` para el campo de texto (por ejemplo adoptando el protocolo `UITextFieldDelegate` en la misma clase `UADetailViewController`). El delegado deberá implementar el método `textField:shouldReturn:`, y en él devolveremos YES para indicar que se debe cerrar el teclado.

b) Vamos a añadir un evento al botón *Guardar* para que almacene en el objeto los nuevos valores que hayamos introducido en los campos de texto. Crea una acción para hacer esto en `UADetailViewController` y conéctala con el evento apropiado del botón. Comprueba ahora que los datos quedan almacenados correctamente cuando pasamos de una película a otra.

c) Si guardamos los datos y volvemos a la lista, veremos que el título no cambia, aunque lo hayamos modificado en la ficha con los datos de la película. Para conseguir que cambie deberemos recargar la tabla que muestra los datos. Una forma sencilla de hacer esto es recargar la tabla cada vez que se muestra la pantalla, implementando el siguiente método en `UAMasterViewController`:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}
```

8.4. Imágenes

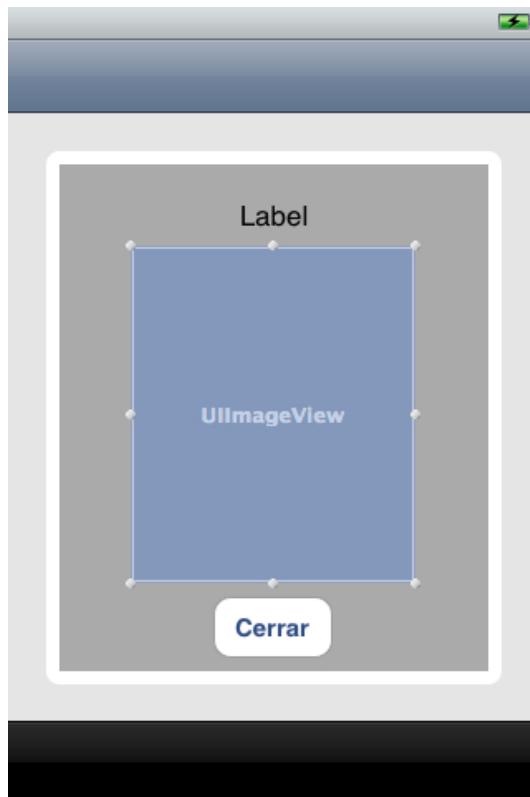
Vamos a añadir imágenes a la aplicación. Para ello puedes utilizar las proporcionadas en las plantillas o descargarlas de Internet. Se pide:

- a) En primer lugar añadiremos una propiedad de tipo `UIImage` a las películas. Al crear la lista de películas, asignaremos a cada película la imagen que corresponda.
- b) Vamos ahora a mostrar en la tabla la imagen asociada a cada película. Utilizaremos para ello la propiedad `imageView.image` del objeto `UITableViewCell` al configurar cada celda.

8.5. Carga de componentes del NIB

Ahora vamos a dotar de funcionalidad el botón *Ver cartel* que tenemos en la pantalla con los datos de las películas. Se pide:

- a) Crea la vista con el cartel de la película en un NIB independiente, con el siguiente aspecto:



Vista del cartel de la película

De este NIB sólo nos interesa el recuadro gris, que contiene como subvistas una etiqueta con el título de la película (`UILabel`), una imagen con su cartel (`UIImageView`), y un botón para cerrar la ventana. Por lo tanto, será este recuadro el que tendremos que conectar con el código de la aplicación mediante un *outlet*.

- b)* Haz que al pulsar el botón *Ver cartel* se cargue la vista anterior del NIB y se muestre en pantalla (añadiéndola como subvista). Deberá quedar por encima de los datos de la película. Para llenar los datos del título de la película y de su imagen puedes asignar *tags* a cada uno de estos componentes, y en el código hacer un `viewWithTag` desde la vista que los contiene.
- c)* Haz que al pulsar el botón *Cerrar* de la subvista con la imagen, ésta se elimine de su supervista.
- d)* Cambia el atributo necesario de `UIImageView` para que la imagen ocupe el máximo espacio posible dentro del área del componente, sin deformarse ni recortarse.

8.6. Almacenamiento (*)

Sobre el ejercicio anterior, vamos a hacer que cada vez que la aplicación pase a segundo plano se almacene la lista de películas actual en disco para así evitar perder los cambios realizados si la aplicación se cerrase. Se pide:

- a)* ¿Podemos guardar la colección de películas directamente en disco mediante el método que tenemos en las colecciones para ello? ¿Por qué? En caso negativo, ¿cómo podríamos guardarlas?
- b)* Implementa en la clase `UAPelicula` el protocolo necesario para almacenar las películas. Para simplificar, por el momento vamos a guardar únicamente el título y el director (el resto de datos los guardaremos más adelante).
- c)* Implementa en `UAMasterViewController` un método `save` que se encargue de guardar la lista de películas utilizando los métodos de serialización definidos en el apartado anterior (podemos guardar toda la colección mediante una única operación de guardado, no hace falta guardar cada película individualmente).
- d)* Haz que cuando la aplicación pase a segundo plano se ejecute el método `save` anterior. Esto deberemos implementarlo en el método adecuado de `UAAppDelegate`. Para poder hacer esto, en el *app delegate* deberemos guardarnos una referencia a `UAMasterViewController` (por ejemplo podemos guardarla como propiedad de la clase). Comprueba que el guardado se realiza correctamente cuando pulsamos el botón HOME del teléfono, por ejemplo escribiendo un *log* que nos indique el resultado de la operación de guardado (YES o NO).
- e)* En el inicializador de la clase `UAMasterViewController` haz que intente leer la colección del fichero en disco. Si el fichero existe, cogerá la colección almacenada en él, si no, inicializará la colección por defecto como hacíamos anteriormente. Comprueba que

si se modifica el título o el director de alguna película la próxima vez que carguemos la aplicación esos cambios se conservan.

f) Vamos a guardar ahora también el resto de campos de las películas. El campo más complejo es la imagen, ya que no es un tipo de datos que podamos guardar directamente. Para solucionar esto, podemos guardar la imagen como un objeto de tipo `NSData` (datos binarios) que contenga su representación en PNG. Podemos obtener dicha representación con:

```
NSData *datos = UIImagePNGRepresentation(self.cartel);
```

Para crear de nuevo una imagen a partir de los datos binarios de su representación en PNG, podemos utilizar el siguiente inicializador:

```
UIImage *imagen = [UIImage imageWithData: datos];
```

8.7. Creación programática de componentes (*)

Vamos a crear componentes de forma programática. Añadiremos a cada elemento de la tabla (en `UIMasterViewController`) un interruptor que nos permitirá cambiar el color de la fila. Se pide:

- a) En el método `tableView:cellForRowAtIndexPath:` de `UAMasterViewController`, tras la instanciación de las celdas, crea también un nuevo objeto de tipo `UISwitch` utilizando el inicializador vacío, y asígnaselo a la propiedad `accesoryView` de la celda.
- b) En el mismo lugar, añade una acción de forma programática al `UISwitch` para que al cambiar de valor cambie el color de fondo y de texto de la celda. Cuando el interruptor esté activado el fondo de la celda será gris claro, y el texto rojo:



Lista de películas con interruptor

9. Controladores

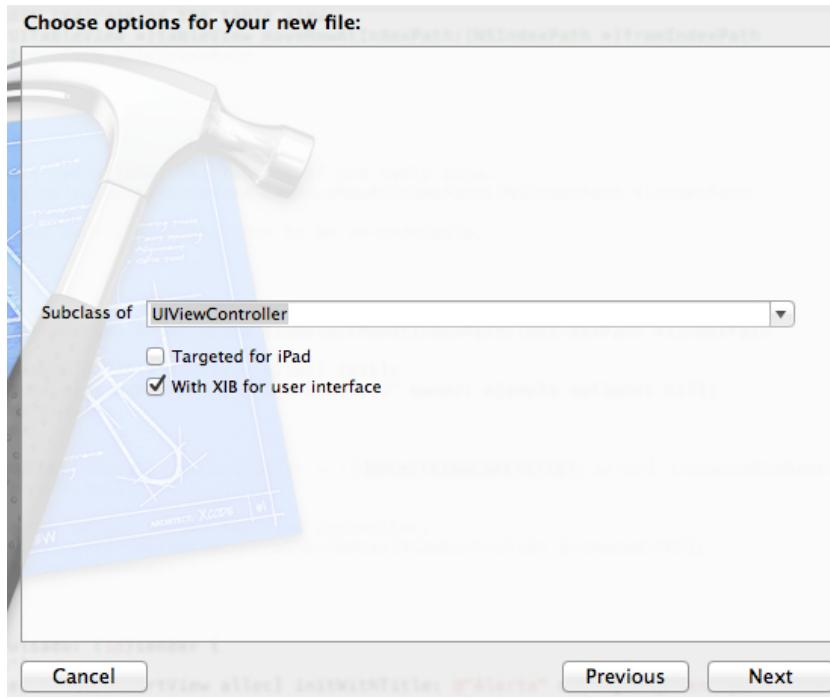
En la sesión anterior hemos visto cómo crear la interfaz con *Interface Builder* y como cargar las vistas definidas en el fichero NIB desde la aplicación. Sin embargo, lo más frecuente será crear un controlador asociado a cada fichero NIB que se encargue de gestionar la vista. Los controladores serán clases que heredarán de `UIViewController`, que tiene una propiedad `view` que se encargará de referenciar y retener la vista que gestiona.

Este controlador se utilizará además como *File's Owner* del NIB. Podríamos utilizar cualquier tipo de objeto como *File's Owner*, pero utilizar un objeto de tipo `UIViewController` nos aportará numerosas ventajas, como por ejemplo incorporar métodos para controlar el ciclo de vida de la vista y responder ante posibles cambios en el dispositivo.

Cuando mostremos en la pantalla la vista asociada al controlador (propiedad `view`), por ejemplo añadiéndola como subvista de la ventana principal, los eventos del ciclo de vida de dicha vista (carga, cambio de orientación, destrucción, etc) pasarán a ser gestionados por nuestro controlador.

9.1. Creación de un controlador propio

Para crear un nuevo controlador seleccionaremos *File > New File ... > iOS > Cocoa Touch > UIViewController subclass*. Nos permitirá crear una subclase de `UIViewController` o de `UITableViewController`, que es un tipo de controlador especializado para la gestión de tablas. También nos permite elegir si queremos crear un controlador destinado al iPad, o si queremos que cree automáticamente un fichero NIB asociado. Tras esto tendremos que introducir el nombre del fichero en el que guardaremos el controlador (que coincidirá con el nombre de la clase y del fichero NIB si hemos optado por crearlo).



Nuevo controlador

Normalmente crearemos el fichero NIB (XIB) junto al controlador. De esta forma ya creará el fichero NIB con el *File's Owner* configurado del tipo del controlador que estamos creando, y con una vista (`UIView`) raíz vinculada con la propiedad `view` del controlador. Dicha propiedad está definida en `UIViewController`, y se utilizará para referenciar la vista raíz de la pantalla. Por ejemplo, si hemos creado un controlador llamado `EjemploViewController` el *File's Owner* del NIB asociado será de dicha clase (atributo `Class` del inspector de identidad).

Podemos cargar un controlador, junto a su vista asociada, de la siguiente forma:

```
EjemploViewController *ejemploViewController =  
    [[EjemploViewController alloc]  
     initWithNibName:@"EjemploViewController" bundle:nil];
```

De esta forma el controlador se encarga de cargar la vista del fichero NIB automáticamente (no tenemos que cargarlo manualmente como hicimos en la sesión anterior). Cuando mostremos la vista de dicho controlador en pantalla (propiedad `ejemploViewController.view`) se irán ejecutando una serie de métodos del controlador que nos permitirán gestionar el ciclo de vida de dicha vista.

Nota

Por defecto Xcode le da el mismo nombre al fichero NIB que al controlador, en nuestro caso `EjemploViewController`. Sin embargo, no parece adecuado ponerle sufijo `Controller` a un fichero que sólo contiene la vista, sería más conveniente llamarlo `EjemploView.xib`. Podemos cambiarle el nombre manualmente. Incluso la API de Cocoa Touch tiene esto en cuenta, y aunque busquemos un NIB con sufijo `Controller`, si no lo encuentra buscará si

existe el fichero sin ese sufijo. De esta forma, al cargar el controlador podríamos no especificar ningún nombre de fichero NIB (pasamos `nil` como parámetro), y como buscará un NIB que se llame como el controlador, debido a la característica que acabamos de comentar será capaz de localizar el NIB aunque no lleve el sufijo `Controller`.

Si queremos que cargue el NIB por defecto, también podemos utilizar simplemente su inicializador `init`, que será equivalente a llamar al inicializador indicado anteriormente (que es el inicializado designado) pasando `nil` a sus dos parámetros.

```
EjemploViewController *ejemploViewController =
    [[EjemploViewController alloc] init];
```

Nota

Si el fichero NIB por defecto asociado al controlador no existiese, el controlador creará automáticamente una vista vacía con el tamaño de la ventana principal (`applicationFrame`).

Una vez inicializado el controlador, podemos hacer que su contenido se muestre en pantalla añadiendo su vista asociada (propiedad `view`) a la ventana principal que esté mostrando la aplicación, o a alguna de sus subvistas.

```
[self.window addSubview: ejemploViewController.view];
```

También deberemos retener el controlador en memoria, en alguna propiedad de nuestra clase, que en el caso anterior hemos supuesto que es *Application Delegate*. Por ese motivo también teníamos una propiedad que hacía referencia a la ventana principal. Si estuviésemos en otra clase, recordamos que podríamos hacer referencia a esta ventana con `[[UIApplication sharedApplication] keyWindow]`.

Atajo

Si nuestra aplicación está destinada sólo a iOS 4.0 o superior, podemos mostrar la vista de nuestro controlador en la ventana principal y retener el controlador en una única operación, sin tener que crear una nueva propiedad para ello. Esto es gracias a la propiedad `rootViewController` de `UIWindow`, a la que podemos añadir directamente el controlador y se encargará de retenerlo y de mostrar su vista en la ventana.

Una vez es mostrada en pantalla la vista, su ciclo de vida comenzará a gestionarse mediante llamadas a métodos del controlador al que está asociada (y que nosotros podemos sobrescribir para dar respuesta a tales eventos).

9.1.1. Ciclo de vida de los controladores

Los métodos básicos que podemos sobrescribir en un controlador para recibir notificaciones del ciclo de vida de su vista asociada son los siguientes:

- `(void)loadView`
- `(void)viewDidLoad`
- `(void)viewDidUnload`

```
- (void)didReceiveMemoryWarning
```

El más importante de estos métodos es `viewDidLoad`. Este método se ejecutará cuando la vista ya se haya cargado. En él podemos inicializar propiedades de la vista, como por ejemplo establecer los textos de los campos de la pantalla:

```
- (void)viewDidLoad {
    descripcionView.text = asignatura.descripcion;
    descripcionView.editable = NO;
}
```

Normalmente no deberemos sobrescribir el método `loadView`, ya que este método es el que realiza la carga de la vista a partir del NIB proporcionado en la inicialización. Sólo lo sobrescribiremos si queremos inicializar la interfaz asociada al controlador de forma programática, en lugar de hacerlo a partir de un NIB. En este método tendremos que crear la vista principal gestionada por el controlador, y todas las subvistas que sean necesarias:

```
- (void)loadView {
    UIView *vista = [[UIView alloc]
                      initWithFrame: [[UIScreen mainScreen] applicationFrame]];
    ...
    self.view = vista;
    [vista release];
}
```

Podríamos también redefinir este método para cargar el contenido de un NIB de forma manual, tal como vimos en la sesión anterior, y asignarlo a la propiedad `view`.

Por último, el método `viewDidUnload` podrá ser llamado en situaciones de escasa memoria en las que se necesite liberar espacio y la vista no esté siendo mostrada en pantalla. Realmente, en situaciones de baja memoria se llamará al método `didReceiveMemoryWarning`. Este método comprobará si la vista no está siendo mostrada en pantalla (es decir, si la propiedad `view` del controlador tiene asignada como supervista `nil`). En caso de no estar mostrándose en pantalla, llamará a `viewDidUnload`, donde deberemos liberar todos los objetos relacionados con la vista (otras vistas asociadas que estemos reteniendo, imágenes, etc) que puedan ser recreados posteriormente.

Cuando la vista se vaya a volver a mostrar, se volverá a llamar a los métodos `loadView` y `viewDidLoad` en los que se volverá a crear e inicializar la vista.

Consejo

Un práctica recomendable es liberar los objetos de la vista en `viewDidUnload` y los objetos del modelo en `didReceiveMemoryWarning`, ya que este segundo podría ejecutarse en ocasiones en las que no se ejecuta el primero (cuando la vista esté siendo mostrada). Si sucede esto, no se volverá a llamar a `loadView` ni `viewDidLoad`, por lo que no podemos confiar en estos métodos para reconstruir los objetos del modelo. Lo recomendable es crear *getters* que comprueben si las propiedades son `nil`, y que en tal caso las reconstruyan (por ejemplo accediendo a la base de datos). Si sobrescribimos `didReceiveMemoryWarning`, no debemos olvidar llamar a `super`, ya que de lo contrario no se hará la llamada a `viewDidUnload` cuando la vista no sea visible.

A parte de estos métodos, encontramos otros métodos que nos avisan de cuándo la vista va a aparecer o desaparecer de pantalla, o cuándo lo ha hecho ya:

```
- viewWillAppears:  
- viewDidAppear:  
- viewWillDisappear:  
- viewDidDisappear:
```

Todos ellos reciben un parámetro indicando si la aparición (desaparición) se hace mediante una animación. Si los sobrescribimos, siempre deberemos llamar al correspondiente super.

9.1.2. Control de la orientación

El controlador también incorpora una serie de métodos para tratar la orientación de la vista. Esto nos facilitará gestionar los cambios de orientación del dispositivo y adaptar la vista a cada caso concreto. El método principal que deberemos sobrescribir si queremos permitir y controlar el cambio de orientación es el siguiente:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:  
    (UIInterfaceOrientation)interfaceOrientation
```

El tipo del parámetro UIInterfaceOrientation es una enumeración con las cuatro posibles orientaciones del dispositivo que reconocemos (vertical hacia arriba, horizontal izquierda, horizontal derecha y vertical hacia abajo). Cuando el usuario cambie la orientación del dispositivo se llamará a este método con la orientación actual para preguntarnos si la soportamos o no. Si devolvemos YES, entonces cambiará la vista para adaptarla a dicha orientación. Podemos utilizar las macros UIInterfaceOrientationIsLandscape() o UIInterfaceOrientationIsPortrait() para comprobar si la orientación es horizontal o vertical independientemente de su sentido.

Ayuda

Podemos probar los cambios de orientación en el simulador seleccionando *Hardware > Girar a la izquierda* (*cmd+Cursor izq.*) o *Hardware > Girar a la derecha* (*cmd+Cursor der.*). Si en el método anterior devolvemos siempre YES, veremos como la vista se adapta a todas las orientaciones posibles.

Si hemos configurado correctamente la vista para que el tamaño se ajuste automáticamente, no debe haber ningún problema al hacer la rotación. De todas formas, en UIViewController tenemos métodos adicionales para controlar la forma en la que se realiza la rotación con más detalle:

- `willRotateToInterfaceOrientation:duration:`: Se va a realizar la rotación, pero todavía no se han actualizado los límites de la pantalla
- `willAnimateRotationToInterfaceOrientation:duration:`: Los límites de la pantalla ya se han actualizado (se ha intercambiado el ancho y el alto), pero todavía no se ha realizado la animación de la rotación.

- `didRotateFromInterfaceOrientation:`: La rotación se ha completado. Se puede utilizar tanto el método anterior como este para ajustar el contenido de la vista al nuevo tamaño de la pantalla.

Nota

La orientación inicial con la que arranca la aplicación podrá ser una de las indicadas en el fichero `Info.plist`. Si es distinta a la orientación vertical, deberemos llevar cuidado al inicializar la vista, ya que en `viewDidLoad` todavía no se habrá ajustado el tamaño correcto de la pantalla. El tamaño correcto lo tendremos cuando se llame a `willAnimateRotationToInterfaceOrientation: duration:` o `didRotateFromInterfaceOrientation:`.

9.2. Controlador para las tablas

9.2.1. Fuente de datos

En el caso de las tablas (`UITableView`), la gestión de los datos que muestra el componente es más compleja. Para llenar los datos de una tabla debemos definir una fuente de datos, que será una clase que implemente el protocolo `UITableViewDataSource`. Este protocolo nos obligará a definir al menos los siguientes métodos:

```
- (NSInteger) tableView:(UITableView *)tabla  
    numberOfRowsInSection: (NSInteger)sección  
- (UITableViewCell *) tableView:(UITableView *)tabla  
    cellForRowAtIndexPath: (NSIndexPath *)indice
```

En el primero de ellos deberemos devolver el número de elementos que vamos a mostrar en la sección de la tabla indicada mediante el parámetro `numberOfRowsInSection`. Si no indicamos lo contrario, por defecto la tabla tendrá una única sección, así que en ese caso podríamos ignorar este parámetro ya que siempre será 0. Podemos especificar un número distinto de secciones si también definimos el método opcional `numberOfSectionsInTableView`, haciendo que devuelva el número de secciones de la tabla.

El segundo es el que realmente proporciona el contenido de la tabla. En él deberemos crear e inicializar cada celda de la tabla, y devolver dicha celda como un objeto de tipo `UITableViewCell`. En el parámetro `cellForRowIndexPath` se nos proporciona el índice de la celda que tendremos que devolver. La creación de las celdas se hará bajo demanda. Es decir, sólo se nos solicitarán las celdas que se estén mostrando en pantalla en un momento dado. Cuando hagamos *scroll* se irán solicitando los nuevos elementos que vayan entrando en pantalla sobre la marcha. Esto podría resultar muy costoso si cada vez que se solicita una celda tuviésemos que crear un nuevo objeto. Por este motivo realmente lo que haremos es reutilizar las celdas que hayan quedado fuera de pantalla.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
```

```

        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // (1) Reutilizamos una celda del pool
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }

    // (2) Introducimos los datos en la celda reutilizada
    cell.textLabel.text =
        [NSString stringWithFormat:@"Item %d", indexPath.row];

    return cell;
}

```

En la primera parte simplemente buscamos una celda disponible en la cola de celdas reutilizables que incorpora el objeto `UITableView`. Si no encuentra ninguna disponible, instancia una nueva y le asigna el identificación de reutilización con el que las estamos referenciando, para que cuando deje de utilizarse pase a la cola de reutilización de la tabla, y pueda reutilizarse en sucesivas llamadas.

Item 0

Item 1

Item 2

Item 3

Item 4

Ejemplo de UITableView

En la segunda parte configuramos la celda para asignarle el contenido y el aspecto que deba tener el elemento en la posición `indexPath.row` de la tabla. En el ejemplo anterior cada elemento sólo tiene una cadena en la que se indica el número de la fila. Una implementación común consiste en mapear a la tabla los datos almacenados en un `NSArray`. Esto se puede hacer de forma sencilla:

```

- (NSInteger) tableView:(UITableView *)tabla
    numberOfRowsInSection: (NSInteger)seccion
{
    return [asignaturas count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

```

```
if (cell == nil) {
    cell = [[[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CellIdentifier] autorelease];
}

// Configuramos la celda
cell.textLabel.text =
    [[asignaturas objectAtIndex:indexPath.row] nombre];

return cell;
}
```

Como vemos, mostrar los elementos de un `NSArray` en una tabla es tan sencillo como devolver el número de elementos del `array` (`count`) en `tableView:numberOfRowsInSection:`, y en `tableView:cellForRowAtIndexPath:` mostrar en la celda el valor del elemento en la posición `indexPath.row` del `array`.

9.2.2. Delegado de la tabla

Al utilizar una tabla normalmente necesitaremos además un objeto delegado que implemente el protocolo `UITableViewDelegate`, que nos permitirá controlar los diferentes eventos de manipulación de la tabla, como por ejemplo seleccionar un elemento, moverlo a otra posición, o borrarlo.

La operación más común del delegado es la de seleccionar un elemento de la tabla. Para ello hay que definir el método `tableView:didSelectRowAtIndexPath:`, que recibirá el índice del elemento seleccionado.

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UAAsignatura *asignatura =
        [asignaturas objectAtIndex:indexPath.row];
    UIAlertView *alert =
        [[UIAlertView alloc] initWithTitle:asignatura.nombre
                                    message:asignatura.descripcion
                                    delegate: nil
                           cancelButtonTitle:@"Cerrar"
                           otherButtonTitles: nil];
    [alert show];
    [alert release];
}
```

En este caso al pulsar sobre un elemento de la tabla, nos aparecerá una alerta que tendrá como título el nombre de la asignatura seleccionada, y como mensaje su descripción, y además un botón que simplemente cerrará la alerta.

Dado que normalmente siempre necesitaremos los dos objetos anteriores (fuente de datos y delegado) al utilizar tablas, para este tipo de componentes tenemos un controlador especial llamado `UITableViewController`, que hereda de `UIViewController`, y además implementa los protocolos `UITableViewDataSource` y `UITableViewDelegate`. En este controlador la propiedad `view` referenciará a una vista de tipo `UITableView`, y esta vista a

su vez tendrá dos *outlets* (*delegate* y *dataSource*) que deberán hacer referencia al controlador, que es quien se comporta a su vez como delegado y fuente de datos.



Conexiones de UITableView

Si cambiamos el conjunto de datos a mostrar en la tabla, para que los cambios se reflejen en la pantalla deberemos llamar al método `reloadData` del objeto `UITableView`.

```
[self.tableView reloadData];
```

9.2.3. Modo de edición

Las tablas cuentan con un modo de edición en el que podemos modificar los elementos de la tabla, cambiarlos de orden, eliminarlos, o añadir nuevos. Para gestionar este modo de edición bastará con añadir algunos métodos adicionales del *data source* y del delegado de la tabla. En el controlador de la tabla (*TableView Controller*) también disponemos de facilidades para trabajar con este modo de edición. Esto es muy útil y bastante intuitivo por ejemplo para aplicaciones de gestión de tareas. Para entender el funcionamiento y aprender a implementar esto vamos a realizar un ejemplo muy sencillo de aplicación en la que podremos eliminar, editar, añadir y reordenar tareas.

Para esta aplicación crearemos primero los métodos vistos anteriormente (`numberOfRowsInSection` y `cellForRowAtIndexPath`) para poblar la tabla de datos:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Iniciamos el array
    self.arrayTareas = [[NSMutableArray alloc] init];

    // Creamos las tareas
    [self.arrayTareas addObject:@"Hacer la compra"];
    [self.arrayTareas addObject:@"Cambiar de seguro de coche"];
    [self.arrayTareas addObject:@"Hacer deporte"];
    [self.arrayTareas addObject:@"Ir al banco"];
    [self.arrayTareas addObject:@"Confirmar asistencia cumple"];
    [self.arrayTareas addObject:@"Llamar a Juan"];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [self.arrayTareas count];
}

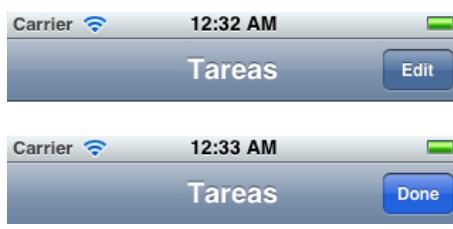
- (UITableViewCell *)tableView:(UITableView *)tableView
```

```
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                  reuseIdentifier:CellIdentifier] autorelease];
    }
    cell.textLabel.text = [self.arrayTareas objectAtIndex:indexPath.row];
    return cell;
}
```

En la plantilla que Xcode crea de un `UITableViewController`, en `viewDidLoad` veremos que aparece comentada una línea que se encarga de añadir un botón de edición a la barra de navegación.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Uncomment the following line to preserve selection
    // between presentations.
    // self.clearsSelectionOnViewWillAppear = NO;
    // Uncomment the following line to display an Edit button in
    // the navigation bar for this view controller.
    self.navigationItem.rightBarButtonItem = self.editButtonItem;
    ...
}
```

Este botón se encargará de alternar entre el modo de edición y el modo normal, cambiando de aspecto según el modo actual.



Botón de edición

Si queremos realizar alguna acción cada vez que cambiamos entre el modo de edición y el modo normal, podemos sobrescribir el método `setEditing:animated:` del controlador:

```
- (void)setEditing:(BOOL)editing animated:(BOOL)animated {
    [super setEditing:editing animated:animated];
}
```

Alternativa. Podemos alternar también manualmente entre el modo de edición y el normal llamando manualmente al método anterior. Por ejemplo, podríamos crear un botón propio que haga esto:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

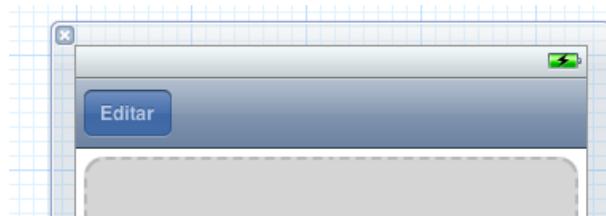
    // Creamos el botón de editar a la izquierda de la barra de navegación
    UIBarButtonItem *botonEditar = [[UIBarButtonItem alloc]
                                    initWithTitle:@"Editar"
                                    style:UIBarButtonItemSystemItemAction
                                    target:self
                                    action:@selector(editarTabla:)];

    self.navigationItem.leftBarButtonItem = botonEditar;
    [botonEditar release];

    ...
}

-(void) editarTabla:(id)sender {
    if (self.editing) {
        [super setEditing:NO animated:YES];
        [self.navigationItem.leftBarButtonItem setTitle:@"Editar"];
        [self.navigationItem.leftBarButtonItem
         setStyle:UIBarButtonItemStylePlain];
    } else {
        [super setEditing:YES animated:YES];
        [self.navigationItem.leftBarButtonItem setTitle:@"Hecho"];
        [self.navigationItem.leftBarButtonItem
         setStyle:UIBarButtonItemStyleDone];
    }
}

```



Botón editar en barra superior

En la implementación anterior hemos realizado de forma manual lo mismo que hace de forma automática el botón `self.editButtonItem` proporcionado por el sistema.

Una vez hecho esto ya podemos probar la aplicación. Veremos que al pulsar el botón de editar éste cambia y se añaden unos pequeños botones a la izquierda de cada una de las celdas, si los pulsamos aparecerá un nuevo botón a la derecha el cual, al pulsarlo deberá borrar la fila. Esto será lo siguiente que implementaremos.

9.2.4. Borrado

Para implementar el borrado de elementos de la tabla debemos descomentar el método `commitEditingStyle` y completarlo con el siguiente fragmento de código:

```

if (editingStyle == UITableViewCellStyleDelete) {
    [self.arrayTareas removeObjectAtIndex:indexPath.row];
    [self.tableView
     deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]

```

```
        withRowAnimation:UITableViewRowAnimationRight];
} else if (editingStyle == UITableViewCellStyleInsert) {
```

Como vemos, con este método indicamos la acción a realizar cuando se realiza alguna acción de edición. Por el momento sólo hemos indicado lo que debe hacer cuando se pulsa sobre el botón de borrar un elemento de la tabla, que es el que aparece por defecto.

Podemos fijarnos también en que en primer lugar debemos eliminar el *item* correspondiente a la fila de nuestro almacen de datos (en nuestro caso el *array* de tareas en memoria). Tras esto, llamamos a `deleteRowsAtIndexPaths:withRowAnimation:` sobre la tabla para que la fila que contenía el dato eliminado desaparezca mediante una animación.

Importante

Al eliminar una fila de la tabla, se comprueba que tras la eliminación el *data source* proporcione un elemento menos que antes. Por lo tanto es fundamental haber borrado el *item* de nuestro *array* de tareas, ya que de no haberlo hecho se producirá un error en tiempo de ejecución y se cerrará la aplicación.

Al implementar `commitEditingStyle` también se habilita el modo *swipe to delete*. Esto consiste en que si hacemos un gesto de barrido horizontal sobre una de las filas de la tabla, nos dará la opción de borrarla.

9.2.5. Reordenación

Dentro del modo de edición también tenemos la posibilidad de reordenar las filas de la tabla, arrastrándolas mediante un ícono que aparece en su parte derecha. Para poder realizar esto debemos implementar los siguientes métodos del delegado de la tabla:

```
// Override to support rearranging the table view.
- (void)tableView:(UITableView *)tableView
    moveRowAtIndexPath:(NSIndexPath *)fromIndexPath
    toIndexPath:(NSIndexPath *)toIndexPath
{
    NSString *item = [self.arrayTareas objectAtIndex:fromIndexPath.row];
    [self.arrayTareas removeObjectAtIndex:fromIndexPath.row];
    [self.arrayTareas insertObject:item atIndex:toIndexPath.row];
}

// Override to support conditional rearranging of the table view.
- (BOOL)tableView:(UITableView *)tableView
    canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Return NO if you do not want the item to be re-orderable.
    return YES;
}
```

Con esto podremos especificar la forma en la que la reordenación de la tabla se refleja en los datos almacenados, y también podemos indicar qué elementos son ordenables y cuáles no lo son.

9.2.6. Inserción

Por último, vamos a ver cómo insertar nuevos elementos en la tabla. Para ello tenemos diferentes opciones, entre las que destacamos:

- Añadir un botón en la barra de navegación que sirva para insertar un nuevo *item* en la tabla. Podemos utilizar el botón proporcionado por el sistema para añadir elementos.
- Añadir una nueva entrada en la tabla sólo cuando estemos en modo de edición, que nos permita añadir un nuevo *item*. Esta entrada tendrá como ícono el símbolo '+', en lugar de '-' como ocurre por defecto para las filas cuyo estilo de edición es borrado.

Si optamos por el botón en la barra de navegación, la inserción de elementos es sencilla. En el evento de pulsación del botón añadiremos el nuevo *item* a nuestro almacén interno de datos, y tras esto añadiremos el nuevo elemento a la tabla mediante una animación:

```
- (void)addTarea:(id)sender {
    NSIndexPath *indexPath =
        [NSIndexPath indexPathForRow:[self.arrayTareas count]
                     inSection:0];
    [self.arrayTareas addObject:@"Tarea nueva"];
    [self.tableView
        insertRowsAtIndexPaths: [NSArray arrayWithObject:indexPath]
        withRowAnimation:UITableViewRowAnimationTop];
}
```

Si optamos por la opción de añadir una fila adicional para la inserción, en primer lugar deberemos hacer que el *data source* proporcione dicha fila en el caso en el que estemos en modo de edición:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    if(self.editing) {
        // Una fila adicional al final de todas para añadir nuevas
filas
        return [self.arrayTareas count]+1;
    } else {
        return [self.arrayTareas count];
    }
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }

    // Configure the cell.
    if(indexPath.row == [self.arrayTareas count]) {
```

```

        // Fila de inserción al final de la tabla
        cell.textLabel.text = @"NUEVA";
    } else {
        cell.textLabel.text =
            [self.arrayTareas objectAtIndex:indexPath.row];
    }
    return cell;
}

```

Además, deberemos hacer que esta fila aparezca o desaparezca cuando entremos y salgamos del modo de edición. Para esto podemos sobrescribir el método que cambia entre el modo de edición y el normal, y según a qué modo estemos cambiando insertar o eliminar mediante una animación la fila de inserción:

```

- (void)setEditing:(BOOL)editing
              animated:(BOOL)animated {
    [super setEditing:editing animated:animated];

    NSArray *rows = [NSArray arrayWithObject:
                     [NSIndexPath indexPathForRow:[self.arrayTareas count]
                           inSection:0]];
    if(editing) {
        [self.tableView insertRowsAtIndexPaths:rows
                                      withRowAnimation:UITableViewRowAnimationTop];
    } else {
        [self.tableView deleteRowsAtIndexPaths:rows
                                      withRowAnimation:UITableViewRowAnimationTop];
    }
}

```

Nota

Al insertar la fila de esta forma, también nos aparecerá cuando entremos en modo *swipe to delete*, ya que este modo también ejecuta el método anterior para hacer que el controlador entre en modo de edición. Podemos saber cuando entramos en modo *swipe to delete* mediante los métodos `tableView:willBeginEditingRowAtIndexPath:` y `tableView:didEndEditingRowAtIndexPath:` del delegado. Podemos implementarlos y utilizarlos para activar y desactivar un *flag* que nos indique si la edición es de tipo *swipe to delete*. En ese caso, podríamos decidir no mostrar la fila de inserción. Otra opción es no mostrar esta fila en `setEditing:animated:`, sino entrar en modo de edición con un botón propio e insertar la fila en el evento de pulsación de dicho botón.

Para que en esta fila aparezca el símbolo de inserción ('+') en lugar de borrado ('-'), deberemos definir el siguiente método del *data source*:

```

- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
    editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
    if(indexPath.row < [self.arrayTareas count]) {
        return UITableViewCellEditingStyleDelete;
    } else {
        return UITableViewCellEditingStyleInsert;
    }
}

```

Para finalizar, haremos que al pulsar sobre el ícono de inserción se realice la acción correspondiente:

```

- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // Delete the row from the data source.
        [self.arrayTareas removeObjectAtIndex:indexPath.row];
        [tableView
            deleteRowsAtIndexPaths: [NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
        // Create a new instance of the appropriate class, insert it
        // into the array, and add a new row to the table view.
        [self.arrayTareas addObject:@"Tarea nueva"];
        [tableView
            insertRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationTop];
    }
}

```

Con esto ya funcionará correctamente la inserción de nuevas tareas. Ahora ya podemos ejecutar nuestro proyecto y comprobar que todo funciona correctamente:

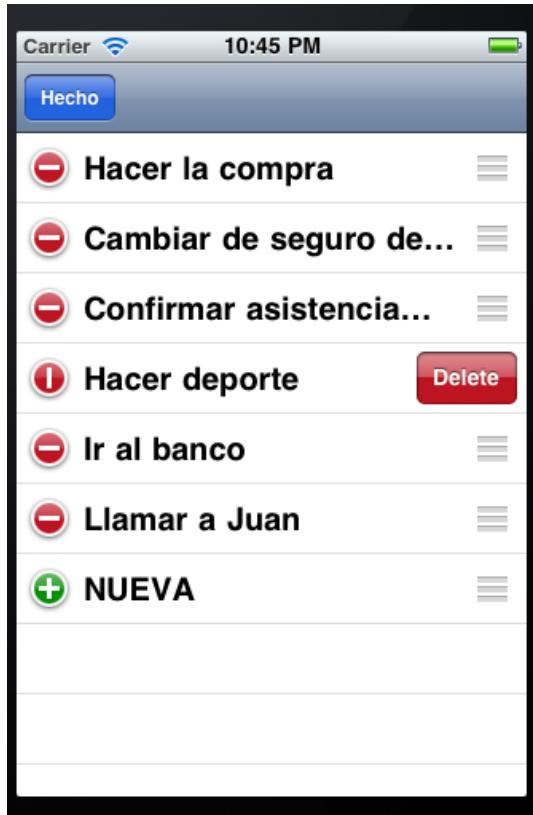


Tabla en modo de edición

Sin embargo, encontramos un problema al combinar la fila de inserción con la reordenación de filas, ya que esta fila no pertenece al *array* de tareas y no puede ser reordenada. Debemos añadir código adicional para evitar este problema. En primer lugar,

haremos que la fila de inserción no sea reordenable:

```
// Override to support conditional rearranging of the table view.  
- (BOOL)tableView:(UITableView *)tableView  
    canMoveRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    // Return NO if you do not want the item to be re-orderable.  
    return indexPath.row < [self.arrayTareas count];  
}
```

Aun así, podría ocurrir que el usuario intentase mover una tarea a una posición posterior a la fila de inserción. Para evitar que esto ocurra definiremos el siguiente método del delegado:

```
- (NSIndexPath *)tableView:(UITableView *)tableView  
targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath *)sourceIndexPath  
toProposedIndexPath:(NSIndexPath *)proposedDestinationIndexPath {  
    if(proposedDestinationIndexPath.row >= [self.arrayTareas count]) {  
        return [NSIndexPath indexPathForRow:[self.peliculas count]-1  
                           inSection:0];  
    } else {  
        return proposedDestinationIndexPath;  
    }  
}
```

Este método nos permite modificar la posición destino en la reordenación de elementos. Cuando el usuario arrastre a una posición, se llama a este método con esa posición como propuesta, pero nosotros podemos cambiarla devolviendo una posición distinta. En nuestro caso, si la posición destino propuesta es posterior a la de la fila especial de inserción, devolvemos como posición de destino definitiva la posición anterior a esta fila, para evitar que la película pueda moverse a una posición inválida. Con esto la aplicación funcionará correctamente.

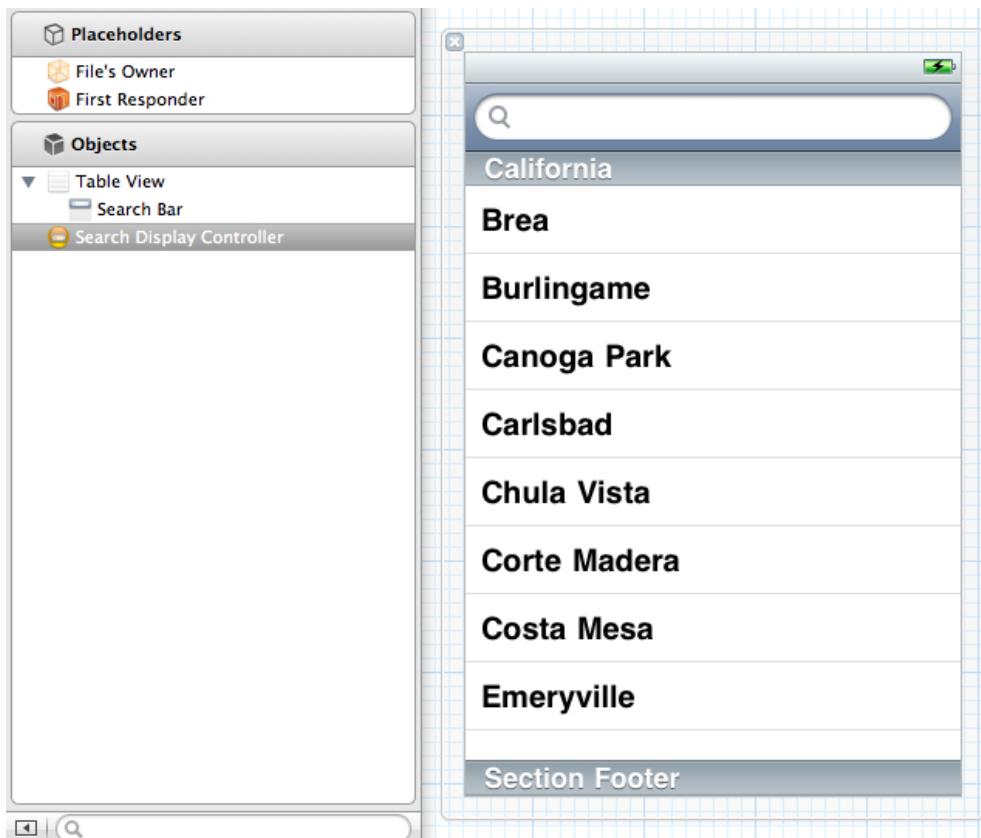
9.3. Controlador de búsqueda

Es común encontrar una barra de búsqueda en las aplicaciones iOS, asociada a una tabla en la que se muestran los resultados de la búsqueda. Este comportamiento estándar se define en el controlador `UISearchDisplayController`. Este controlador es más complejo que los anteriores, por la coordinación necesaria entre los distintos elementos de la interfaz que intervienen en la búsqueda.

Los principales elementos que deberemos proporcionar al controlador de búsqueda son:

- **Barra de búsqueda:** Es la barra de búsqueda en la que el usuario introduce el texto a buscar, y se define con la clase `UISearchBar`. Se guardará en la propiedad `searchBar` del controlador de búsqueda.
- **Controlador de contenido:** Es el controlador en el que se mostrará el contenido resultante de la búsqueda. Normalmente este controlador será de tipo `UITableViewController`, y tendrá asociada una vista de tipo tabla donde se mostrarán los resultados. Se guardará en la propiedad `searchContentsController` del controlador de búsqueda.

Habitualmente la barra de búsqueda se incluye como cabecera de la tabla, y cuando realizamos una búsqueda, permanecerá fija en la parte superior de la pantalla indicando el criterio con el que estamos filtrando los datos actualmente. Este comportamiento es el que se produce por defecto cuando arrastramos un elemento de tipo *Search Bar* sobre una vista de tipo *Table View*. Como alternativa, podemos también arrastrar sobre ella *Search Bar and Search Display Controller*, para que además de la barra nos cree el controlador de búsqueda en el NIB. En ese caso, el controlador de búsqueda será accesible mediante la propiedad `searchDisplayController` de nuestro controlador (podemos ver esto en los *outlets* generados).



Barra de búsqueda en Interface Builder

Si queremos hacer esto mismo de forma programática, en lugar de utilizar Interface Builer, podemos incluir un código como el siguiente:

```
UISearchBar *searchBar = [[UISearchBar alloc] init];
[searchBar sizeToFit];
searchBar.delegate = self;
self.tableView.tableHeaderView = searchBar;

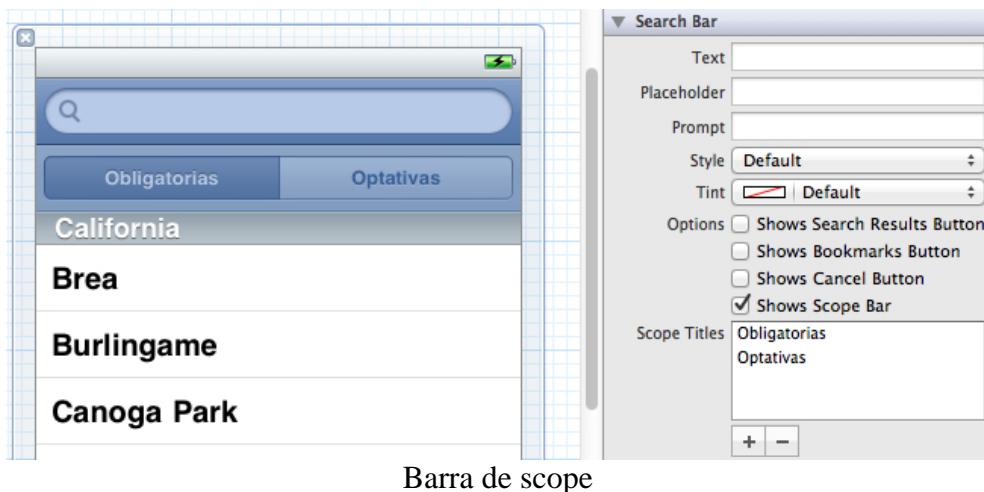
UISearchDisplayController *searchController =
    [[UISearchDisplayController alloc]
        initWithSearchBar:searchBar contentsController:self];
searchController.delegate = self;
```

```
searchController.searchResultsDataSource = self;
searchController.searchResultsDelegate = self;
self.searchController = searchController;
```

Cuidado

Aunque según la documentación de Apple la política de la propiedad `searchDisplayController` es `retain`, al crear el controlador de forma programática se asigna pero no se retiene, por lo que deberemos utilizar una propiedad creada por nosotros para que funcione correctamente. La propiedad `searchDisplayController` sólo nos servirá cuando hayamos creado el controlador mediante Interface Builder.

En la barra podemos introducir un texto que indique al usuario lo que debe introducir (*Placeholder*). De forma opcional, podemos añadir una barra de ámbito (*scope bar*) en la que indicamos el criterio utilizado en la búsqueda. Para ello activaremos esta barra en el inspector de atributos e introduciremos el nombre de cada elemento.



Barra de scope

Es también habitual que la barra de búsqueda permanezca inicialmente oculta, y que tengamos que tirar hacia abajo de la lista para que aparezca. Esto podemos conseguirlo haciendo *scroll* hasta la primera posición de la vista en `viewDidLoad`:

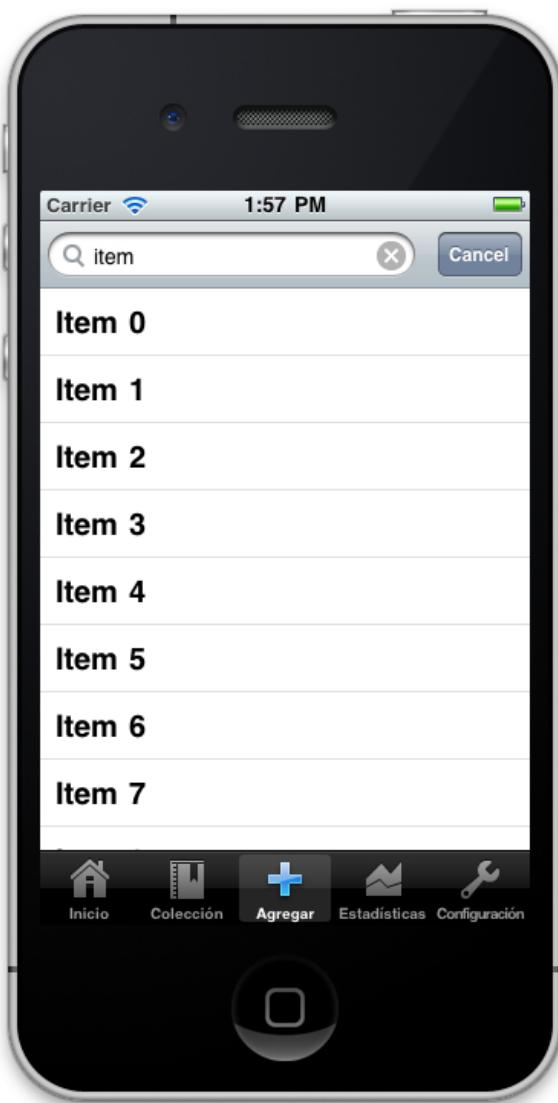
```
[self.tableView
    scrollToRowAtIndexPath: [NSIndexPath indexPathForRow:0
                                         inSection:0]
    atScrollPosition:UITableViewScrollPositionTop
    animated:NO];
```

Cuando el usuario pulsa sobre la barra de búsqueda, el controlador de búsqueda entra en acción, situando la barra de búsqueda en una posición fija en la parte superior de la pantalla, y sombreando el resto de contenido, para que así el usuario centre su atención en llenar el campo de búsqueda.



Foco en la barra de búsqueda

Una vez introducida la cadena de búsqueda, la barra quedará fija en la parte superior de la pantalla, indicando así que lo que estamos viendo son los resultados de la búsqueda, no la tabla original. Podemos volver a la tabla original pulsando sobre el botón *Cancelar*.



Modo de filtrado

Hemos de destacar que realmente existen dos tablas distintas:

- **Tabla original:** Es la tabla que nosotros hemos creado en el NIB, y contiene la colección de datos completa. La encontramos en la propiedad `tableView` de nuestro controlador (`UITableViewController`), al igual que en cualquier tabla.
- **Tabla de resultados:** Es la tabla que crea el controlador de búsqueda para mostrar los resultados producidos por la búsqueda. Se encuentra en la propiedad `searchResultsTableView` del controlador de búsqueda. El controlador de encargará de crearla automáticamente.

Entonces, ¿por qué por defecto estamos viendo los mismos datos en ambos casos? Esto se

debe a que nuestro controlador se está comportando como delegado y fuente de datos de ambas tablas, por lo que resultan idénticas en contenido, aunque sean tablas distintas. Podemos ver en los *outlets* del controlador de búsqueda, que nuestro controlador (*File's Owner*), se comporta como `searchResultsDatasource` y `searchResultsDelegate` (además de ser `dataSource` y `delegate` de la tabla original).

La forma de determinar en la fuente de datos si debemos mostrar los datos originales y los resultados de la búsqueda, será comprobar qué tabla está solicitando los datos:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        return 1;
    } else {
        return [_items count];
    }
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }

    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        cell.textLabel.text = @"Item resultado";
    } else {
        cell.textLabel.text =
            [_items objectAtIndex: indexPath.row];
    }

    return cell;
}
```

La tabla de resultados aparecerá en cuanto introduzcamos texto en el campo de búsqueda. Si hacemos esto en el ejemplo anterior, veremos una tabla como la siguiente:



Resultados de la búsqueda

Sin embargo, lo normal será que cuando se introduzca un texto se realice un filtrado de los datos y esos datos filtrados sean los que se muestren en la tabla de resultado. ¿Cuándo deberemos realizar dicho filtrado? Lo ideal será hacerlo cuando el usuario introduce texto, o bien cuando pulsa el botón *Search*. Podremos estar al tanto de estos eventos de forma sencilla, ya que nuestro controlador, además de todo lo anterior, se comporta también como delegado (campo `delegate`), tanto de la barra de búsqueda (`UISearchBarDelegate`), como del controlador de búsqueda (`UISearchDisplayDelegate`).

Lo habitual será utilizar el método

searchDisplayController:shouldReloadTableForSearchString: de UISearchBarDelegate para realizar la búsqueda. Este método se ejecutará cada vez que cambie la cadena de búsqueda (al teclear en el campo). Deberemos devolver YES si con la cadena introducida queremos que se recargue la vista de resultados, o NO en caso contrario. Por ejemplo, podemos hacer que sólo se realice la búsqueda cuando la cadena introducida tenga una longitud mínima, para así no obtener demasiados resultados cuando hayamos introducido sólo unos pocos caracteres.

```
- (BOOL)searchDisplayController:
    (UISearchDisplayController *)controller
shouldReloadTableForSearchString:(NSString *)searchString {
    self.itemsFiltrados = [self filtrarItems: _items
                                         busqueda: searchString];
    return YES;
}
```

Si hemos incluido una barra de ámbito, podemos responder a cambios del ámbito de forma similar, con el método searchDisplayController:shouldReloadTableForSearchScope:. Es recomendable definir un método genérico para realizar el filtrado que pueda ser utilizado desde los dos eventos anteriores.

Nota

En los métodos anteriores sólo deberemos devolver YES si hemos actualizado la lista de items dentro del propio método. Si queremos hacer una búsqueda en segundo plano, deberemos devolver NO y actualizar la lista una vez obtenidos los resultados.

En algunos casos puede que realizar la búsqueda sea demasiado costoso, y nos puede interesar que sólo se inicie tras pulsar el botón *Search*. Para ello podemos definir el método searchBarSearchButtonClicked: de UISearchBarDelegate:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
    NSString *cadBusqueda = searchBar.text;
    self.itemsFiltrados = [self filtrarItems: _items
                                         busqueda: cadBusqueda];
}
```

Filtraremos los elementos de la vista según la cadena introducida (propiedad `text` de la barra de búsqueda). Guardamos el resultado del filtrado en una propiedad del controlador, y en caso de que se estén mostrando los resultados de la búsqueda, mostraremos dicha lista de elementos:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        return [_itemsFiltrados count];
    } else {
        return [_items count];
}}
```

```
        }
    }

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }

    if(tableView ==
        self.searchDisplayController.searchResultsTableView) {
        cell.textLabel.text =
            [_itemsFiltrados objectAtIndex: indexPath.row];
    } else {
        cell.textLabel.text =
            [_items objectAtIndex: indexPath.row];
    }

    return cell;
}
```

También podemos saber cuándo cambia el texto de la barra de búsqueda con el método `searchBar:textDidChange:`, o cuando cambia el ámbito seleccionado con `searchBar:selectedScopeButtonIndexDidChange:`.

10. Ejercicios de controladores

10.1. Creación de un controlador propio

Vamos a hacer que la pantalla con la imagen del cartel de la película pase a estar gestionada por un controlador. Se pide:

a) Crea un controlador `UACartelViewController`, y haz que aparezca mediante una animación desde la parte inferior de la pantalla cuando pulsemos sobre el botón *Ver Cartel*. Para hacer esto, en el evento del botón podemos introducir el siguiente código:

```
[self presentModalViewController:cartelController animated:YES];
```

b) La pantalla del nuevo controlador debe tener:

- Una barra de navegación que incluirá como elemento derecho un botón para cerrar la vista modal y como título el título de la película.
- Una imagen centrada en la pantalla como único contenido, en la que mostraremos el cartel de la película.

Introduce los elementos anteriores mediante Interface Builder, y conéctalos con el código mediante *outlets*.

c) Haz que al pulsar el botón *Cerrar* se oculte la pantalla mediante una animación. El código para cerrar el controlador será el siguiente:

```
[self dismissModalViewControllerAnimated: YES];
```

d) Permitir que se gire la pantalla, al menos en vertical y horizontal. Adaptar la imagen de forma que siempre aparezcan de forma correcta (debe aparecer centrada en la horizontal, y no debe salirse de la pantalla).



Vista modal en horizontal



Vista modal en vertical

Ayuda

Para que el giro se realice de forma correcta, en primer lugar deberemos permitir distintas orientaciones desde el controlador (modificar para ello el método adecuado de `UACartelViewController`). También deberemos configurar la opciones de autoredimensionamiento de la vista de la imagen (`UIImageView`), para que se adapte al tamaño de la pantalla en la horizontal y en la vertical, y por último deberemos configurar esta vista para que la imagen se muestre con la relación de aspecto adecuada, sin recortar la imagen ni deformarla. Toda la configuración de la vista de la imagen la podremos hacer desde Interface Builder, mediante los atributos que encontraremos en el panel derecho.

10.2. Edición de tablas

Vamos a permitir que el usuario edite la lista de películas, pudiendo borrar las películas existentes, añadir nuevas, o reordenarlas. Para ello deberemos:

- a) Cambiar la propiedad con la lista de películas de tipo `NSArray` a `NSMutableArray`.
- b) Añadir el botón de edición del sistema como botón derecho a la barra de navegación (añadir la línea adecuada en el método `viewDidLoad`).
- c) Permitir borrar películas. Cuando se realice la acción de borrado deberemos borrar la película del *array* y eliminarla de la lista mediante una animación.
- d) Permitir reordenar las películas.
- e) Cuando entremos en modo de edición, añadir una fila adicional para insertar una nueva película. El ícono de acción de la fila debe ser *insertar* en lugar de *borrar*.
- f) Hacer que al realizar la acción de inserción se inserte una nueva película vacía en la última posición de la lista.
- g) Haz que la fila de inserción no pueda reordenarse (debe estar siempre en la última posición). Consigue también que esta fila no aparezca en el modo *swipe to delete*.

10.3. Búsqueda (*)

Implementar una pantalla de búsqueda de películas. Por defecto nos mostrará una lista con todas las películas. Al introducir texto en el cuadro de búsqueda nos aparecerá una lista filtrada por título. Se pide:

- a) Crear un controlador `UABusquedaViewController` de tipo `UITableViewController`. En el `app delegate` haremos que sea este el controlador a cargar como controlador inicial, en lugar de `UAMasterViewController`.
- b) Crear dos propiedades en el controlador `UABusquedaViewController`: `peliculas` y `peliculasFiltradas`, donde tendremos una lista completa de películas, y la lista de películas filtrada según el criterio de búsqueda, respectivamente. Puedes inicializar la lista de películas de la misma forma que en el controlador `UAMasterViewController`.
- c) Muestra la lista de películas anterior en la tabla de `UABusquedaViewController`, igual que hemos hecho anteriormente para `UAMasterViewController`. Comprueba que la lista se visualiza correctamente.
- d) Añade un *search bar* con *search controller* desde Interfaz Builder, y comprueba que el campo de búsqueda aparece correctamente.

Cuidado

Si se inicializa el controlador de la tabla con `initWithStyle`: no cargará los componentes del NIB, sino que creará un `UITableView` de forma automática, por lo que en ese caso no veremos el controlador de búsqueda definido en el NIB. Utiliza el inicializador `initWithNibName:bundle:` para evitar este problema

- e) Haz que en el caso de estar mostrando la tabla de resultados de la búsqueda muestre una única fila con el texto *Buscando*
- f) Haz que al introducir texto en el cuadro de búsqueda se obtenga un listado de las películas filtradas según la cadena de búsqueda indicada. Podemos utilizar el siguiente código:

```
NSMutableArray *seleccionadas = [NSMutableArray
    arrayWithCapacity: [self.peliculas count]];
for(UAPelicula *pelicula in self.peliculas) {
    NSRange rango = [pelicula.titulo rangeOfString: searchString
        options: NSCaseInsensitiveSearch];
    if(rango.location != NSNotFound) {
        [seleccionadas addObject: pelicula];
    }
}
self.peliculasFiltradas = seleccionadas;
```

g) Para terminar, haz que en caso de estar en modo de búsqueda, en lugar de mostrar la fila *Buscando ...*, se muestre la lista de películas filtradas obtenida en el paso anterior. Comprueba que el filtrado funciona correctamente.

11. Transiciones y storyboards

Como hemos comentado, normalmente tendremos un controlador por cada pantalla de la aplicación. Sin embargo, esto no quiere decir que en un momento dado sólo pueda haber un controlador activo, sino que podremos tener controladores dentro de otros controladores. Es muy frecuente encontrar un controlador que se encarga de la navegación (se encarga de mostrar una barra de navegación con el título de la pantalla actual y un botón para volver a la pantalla anterior), que contiene otro controlador que se encarga de gestionar la pantalla por la que estamos navegando actualmente. También tenemos otro tipo de controlador contenedor que se encarga de mostrar las diferentes pantallas de nuestra aplicación en forma de pestañas (*tabs*).

Otra forma de contención se da cuando mostramos una vista modal. Utilizaremos nuestro controlador para mostrar la vista modal, que a su vez estará gestionada por otro controlador.

11.1. Controladores modales

Los controladores modales son la forma más sencilla de contención. Cuando mostramos un controlador de forma modal, su contenido reemplazará al del contenedor actual. Para hacer que un controlador muestre de forma modal otro controlador, llamaremos a su método `presentModalViewController:animated:`:

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UAAsignatura *asignatura =
        [asignaturas objectAtIndex: indexPath.row];

    DetallesViewController *controladorModal =
        [[DetallesViewController alloc]
            initWithAsignatura: asignatura];

    [self presentModalViewController: controladorModal
        animated: YES];

    [controladorModal release];
}
```

En el ejemplo anterior tenemos un controlador de una tabla, en el que al seleccionar un elemento muestra un controlador de forma modal con los detalles del elemento seleccionado. Al presentar el controlador modal, éste se almacena en la propiedad `modalViewController` del controlador padre (la tabla en nuestro ejemplo). Por otro lado, en el controlador modal (detalles en el ejemplo) habrá una propiedad `parentViewController` que hará referencia al controlador padre (tabla en el ejemplo). El controlador modal es retido automáticamente por el controlador padre cuando lo presentamos.

Nota

En el iPhone/iPod touch el controlador modal siempre ocupará toda la pantalla. Sin embargo, en el iPad podemos hacer que esto no sea así. Podemos cambiar esto con la propiedad `modalPresentationStyle` del controlador modal.

Hemos de remarcar que un controlador modal no tiene porque ser un controlador secundario. Se puede utilizar este mecanismo para hacer transiciones a pantallas que pueden ser tan importantes como la del controlador padre, y su contenido puede ser de gran complejidad. Cualquier controlador puede presentar otro controlador de forma modal, incluso podemos presentar controladores modales desde otros controladores modales, creando así una cadena.

Cuando queramos cerrar la vista modal, deberemos llamar al método `dismissModalViewControllerAnimated:` del padre. Sin embargo, si llamamos a este método desde el controlador modal también funcionará, ya que hará un *forward* del mensaje a su padre (propiedad `parentViewController`).

La llamada a `dismissModalViewControllerAnimated:` funcionará de la siguiente forma según sobre qué controlador la llamemos:

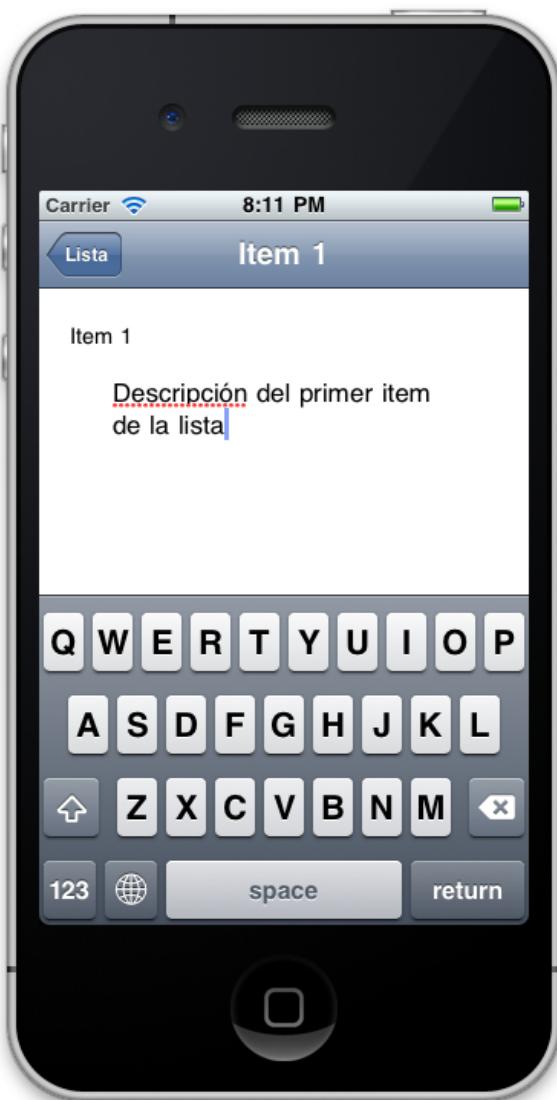
- Controlador con `modalViewController==nil` (el controlador modal que se está mostrando actualmente): Se redirige la llamada a su propiedad `parentViewController`.
- Controlador con `modalViewController!=nil` (controlador que ha presentado un controlador modal hijo): Cierra su controlador modal hijo, y todos los descendientes que este último tenga. Cuando se cierra con una única operación toda la cadena de controladores modales, veremos una única transición de la vista modal que se estuviese mostrando a la vista padre que ha recibido el mensaje para cerrar la vista modal.

11.2. Controlador de navegación

El tipo de aplicación más común que encontramos en iOS es el de las aplicaciones basadas en navegación. En ellas al pasar a una nueva pantalla, ésta se apila sobre la actual, creando así una pila de pantallas por las que hemos pasado. Tenemos una barra de navegación en la parte superior, con el título de la pantalla actual y un botón que nos permite volver a la pantalla anterior.



Pantalla raíz de la navegación

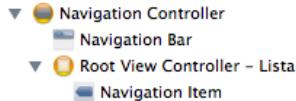


Pantalla de detalles en la pila de navegación

Para conseguir este funcionamiento, contamos con la clase `UINavigationController` que ya implementa este tipo de navegación y los elementos anteriores. Dentro de este controlador tenemos una vista central que es donde se mostrará el contenido de cada pantalla por la que estemos navegando. Cada una de estas pantallas se implementará mediante un controlador.

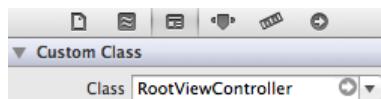
No será necesario crear ninguna subclase de `UINavigationController`, ya que en ella contamos ya con todos los elementos necesarios para implementar este esquema de navegación, pero si que tendremos que tener en cuenta que estamos utilizando este controlador contenedor en los controladores que implementemos para cada pantalla.

Cuando arrastramos un objeto de tipo `UINavigationController` a nuestro fichero NIB, veremos que crear bajo él una estructura con los siguientes elementos:



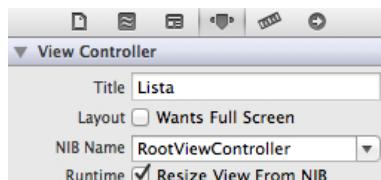
Elementos de Navigation Controller

- *Navigation Bar*: Define el fondo de la barra de navegación en la parte superior de la pantalla.
- *Navigation Item*: Define el contenido que se mostrará en la barra de navegación.
- *Root View Controller*: Controlador raíz que se utilizará para mostrar el contenido de la vista central. Este controlador podrá ser de un tipo propio en el que definamos el contenido de la pantalla raíz de la navegación. Para definir el tipo del controlador utilizaremos la propiedad *Class* del inspector de identidad.



Inspector de identidad del controlador raíz

Si queremos que cargue dicho controlador con un NIB distinto al NIB por defecto, podemos especificar el nombre del NIB en el inspector de atributos.



Inspector de atributos del controlador raíz

Atención

Si nuestro controlador (*Root View Controller* en el caso anterior) se crea dentro del NIB, nunca se llamará a su inicializador designado, ya que los objetos del NIB son objetos ya construidos que cargamos en la aplicación. Por lo tanto, si queremos realizar alguna inicialización, deberemos utilizar dentro de él el método `awakeFromNib`.

De forma alternativa, también podemos crear nuestro `UINavigationController` de forma programática. Para ello en primer lugar crearemos el controlador raíz, y después inicializaremos el controlador de navegación proporcionando en su constructor dicho controlador raíz:

```

RootViewController *rootViewController =
    [[RootViewController alloc] initWithNibName:@"RootViewController"
                                         bundle:nil];
UINavigationController *navController =
    [[UINavigationController alloc]
        initWithRootViewController:rootViewController];
  
```

Vamos a centrarnos en ver cómo implementar estos controladores propios que mostraremos dentro de la navegación.

En primer lugar, `UIViewController` tiene una propiedad `title` que el controlador de navegación utilizará para mostrar el título de la pantalla actual en la barra de navegación. Cuando utilicemos este esquema de navegación deberemos siempre asignar un título a nuestros controladores. Cuando apilemos una nueva pantalla, el título de la pantalla anterior se mostrará como texto del botón de volver atrás.

Además, cuando un controlador se muestre dentro de un controlador de navegación, podremos acceder a dicho controlador contenedor de navegación mediante la propiedad `navigationController` de nuestro controlador.

Nota

Podemos personalizar con mayor detalle los elementos mostrados en la barra de navegación mediante la propiedad `navigationItem` de nuestro controlador. Esta propiedad hace referencia al objeto que veíamos como *Navigation Item* en Interface Builder. Los elementos que podremos añadir a dicha barra serán de tipo *Bar Button Item* (`UIBarButtonItem`). En Interface Builder podemos añadirlos como hijos de *Navigation Item* y conectarlos con *outlets* de dicho elemento.

Cuando queramos pasar a la siguiente pantalla dentro de un controlador de navegación, utilizaremos el método `pushViewController:animated:` del controlador de navegación:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    DetailViewController *detailViewController =
        [[DetailViewController alloc]
            initWithNibName:@"DetailViewController"
            bundle:nil];
    NSString *titulo = [NSString
        stringWithFormat:@"Item %d", indexPath.row];
    detailViewController.title = titulo;

    [self.navigationController
        pushViewController:detailViewController animated:YES];

    [detailViewController release];
}
```

Tras esta llamada se apila, se retiene, y se muestra la siguiente pantalla, que pasa a ser la cima de la pila. En la nueva pantalla veremos un botón para volver atrás en la barra de navegación. Al pulsarlo se desapilará la pantalla y se liberará la referencia. También podemos desapilar las pantallas de forma programática con `popViewControllerAnimated:`, que desapilará la pantalla de la cima de la pila. De forma alternativa, podemos utilizar `popToRootViewControllerAnimated:` o `popToViewController:animated:` para desapilar todas las pantallas hasta llegar a la raíz, o bien hasta llegar a la pantalla especificada, respectivamente.

11.3. Controlador de barra de pestañas

Otro tipo de controlador contenedor predefinido que podemos encontrar es el controlador que nos permite organizar las pantallas de nuestra aplicación en forma de pestañas (*tabs*) en la parte inferior de la pantalla:



Aspecto de un Tab Bar Controller

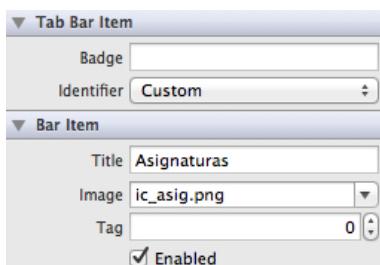
Dicho controlador se implementa en `UITabBarController`, y al igual que en el caso anterior, tampoco deberemos crear subclases de él. Cuando añadamos un elemento de tipo *Tab Bar Controller* a nuestro NIB se creará una estructura como la siguiente:



Elementos de Tab Bar Controller

- *Tab Bar*: Representa la barra de pestañas que aparecerá en la parte inferior de la pantalla, y que contendrá una serie de elementos.
- *Tab Bar Item*: Representa cada uno de los elementos de la barra de pestañas. Habrá un *item* por cada pestaña que tengamos en la barra. Aquí configuraremos el título y el ícono que se mostrará en la pestaña.
- *View Controller*: Tendremos tantos controladores como pestañas en la barra. Podemos añadir nuevos controladores como hijos de *Tab Bar Controller* para añadir nuevas pestañas. Cada controlador tendrá asociado un *Tab Bar Item* con la configuración de su correspondiente pestaña.

Para configurar el aspecto de cada pestaña en la barra, seleccionaremos en el *dock* el correspondiente *Tab Bar Item* y accederemos a su inspector de atributos.



Inspector de atributos de Tab Bar Item

Como vemos, podemos o bien utilizar un aspecto predefinido (propiedad *Identifier*), o bien poner un título (*Title*) e imagen (*Image*) propios.

Iconos de las pestañas

Como iconos para las pestañas deberemos utilizar imágenes PNG de 30 x 30 px con transparencia. Los colores de la imagen (RGB) se ignorarán, y para dibujar el ícono sólo se tendrá en cuenta la capa *alpha* (transparencia).

Podremos utilizar para cada pestaña cualquier tipo de controlador, que definirá el contenido a mostrar en dicha pestaña. Para especificar el tipo utilizaremos la propiedad *Class* de dicho controlador en el inspector de identidad (al igual que en el caso del controlador de navegación). Incluso podríamos poner como controlador para una pestaña un controlador de navegación, de forma que cada pestaña podría contener su propia pila de navegación.

Nota

Si abrimos un controlador modal desde una pestaña o desde una pantalla de navegación, perderemos las pestañas y la barra de navegación.

Podemos crear también este controlador de forma programática. En este caso, tras instanciar el controlador de pestañas, tendremos que asignar a su propiedad `viewController` un *array* de los controladores a utilizar como pestañas. Tendrá tantas pestañas como elementos tenga la lista proporcionada:

```
PrimerViewController *controller1 = [[PrimerViewController alloc]
    initWithNibName:@"PrimerViewController" bundle:nil];
SegundoViewController *controller2 = [[SegundoViewController alloc]
    initWithNibName:@"SegundoViewController" bundle:nil];
TercerViewController *controller3 = [[TercerViewController alloc]
    initWithNibName:@"TercerViewController" bundle:nil];

UITabBarController *tabBarController =
    [[UITabBarController alloc] init];
tabBarController.viewControllers = [NSArray arrayWithObjects:
    controller1, controller2, controller3, nil];
```

El título de cada pestaña y el ícono de la misma se configurarán en cada uno de los controladores incluidos. Por ejemplo, en `PrimerViewController` podríamos tener:

```
- (id)initWithNibName:(NSString *)NibNameOrNil
    bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNilOrNil
        bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
        self.title = @"Asignaturas";
        self.tabBarItem.image = [UIImage imageNamed:@"icono_asig"];
    }
    return self;
}
```

11.4. Uso de storyboards

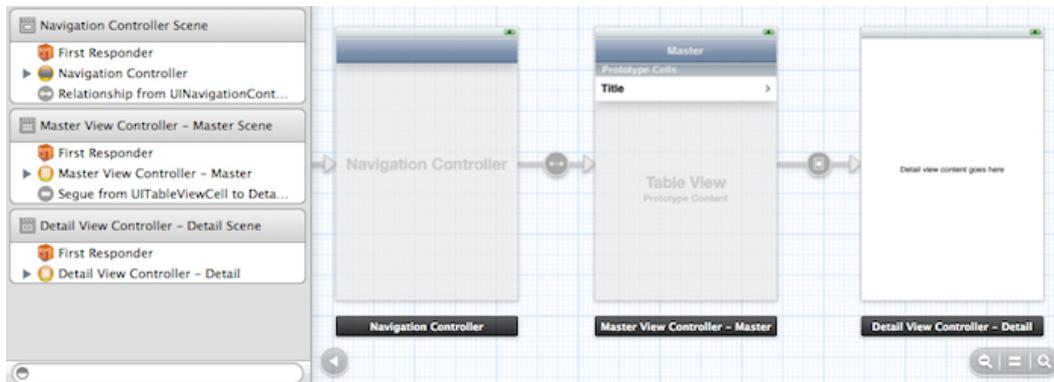
Los *storyboards* son una característica aparecida en iOS 5.0 y Xcode 4.2, que nos dará una forma alternativa para crear la interfaz de nuestras aplicaciones. Nos permitirán crear en un único fichero de forma visual las diferentes pantallas de la aplicación y las transiciones entre ellas. Están pensados principalmente para definir el flujo de trabajo de aplicaciones basadas en navegación o en pestañas.

Advertencia

Los *storyboards* no son compatibles con versiones de iOS previas a la 5.0.

Se definen en ficheros con extensión `.Storyboard`, y podemos crearlos desde Xcode con la opción *New File ... > User Interface > Storyboard*. Por convenio, el *Storyboard* principal de la aplicación se llamará `MainStoryboard.storyboard`.

Para que la aplicación arranque con el contenido del *storyboard*, deberemos especificar dicho fichero en la propiedad *Main Storyboard* del proyecto. Además, en `main.m` deberemos especificar como cuarto parámetro de `UIApplicationMain` la clase delegada de la aplicación, y en dicha clase delegada deberemos tener una propiedad de tipo `UIWindow` llamada `window`.



Editor del storyboard

Sobre el *storyboard* podremos arrastrar diferentes tipos de controladores. Cada uno de estos controladores aparecerán como una pantalla de la aplicación. Para cada controlador deberemos especificar su tipo en el atributo *Class* del inspector de identidad (podrán ser tipos definidos en Cocoa Touch, como `UINavigationController`, o tipos propios derivados de `UIViewController`). En las pantallas del *storyboard* no existe el concepto de *File's Owner*, sino que la relación de las pantallas con el código de nuestra aplicación se establecerá al especificar el tipo de cada una de ellas.

11.4.1. Segues

Las relaciones entre las distintas pantallas se definen mediante los denominados *segues*. Podemos ver los *segues* disponibles en el inspector de conexiones. Encontramos diferentes tipos de *segues*:

- *Relationship*: Define una relación entre dos controladores, no una transición. Nos sirve para relacionar un controlador contenedor con el controlador contenido en él.
- *Modal*: Establece una transición modal a otra pantalla (controlador).
- *Push*: Establece una transición de navegación a otra pantalla, que podrá utilizarse cuando la pantalla principal esté relacionada con un controlador de tipo `UINavigationController`.
- *Custom*: Permite definir una relación que podremos personalizar en el código.

Por ejemplo, si en una pantalla tenemos una lista con varias celdas, podemos conectar mediante un *segue* una de las celdas a un controlador correspondiente a otra pantalla. De esa forma, cuando el usuario pulse sobre dicha celda, se hará una transición a la pantalla con la que conecta el *segue*. Normalmente el origen de un *segue* será un botón o la celda de una tabla, y el destino será un controlador (es decir, la pantalla a la que vamos a hacer

la transición).

En el primer controlador que arrastremos sobre el *storyboard* veremos que aparece una flecha entrante. Este es un *segue* que no tienen ningún nodo de origen, y que indica que es la primera pantalla del *storyboard*.

Si al arrancar la aplicación queremos tener acceso a dicho controlador raíz, por ejemplo para proporcionale datos con los que inicializarse, podemos acceder a él a través de la propiedad `rootViewController` de la ventana principal:

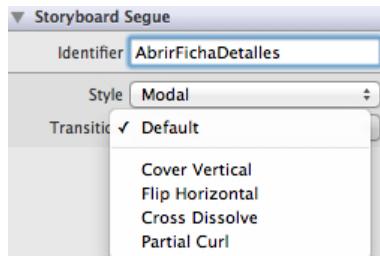
```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UIViewController *controlador =
        (UIViewController *)self.window.rootViewController;
    ...
    return YES;
}
```

Cuando se ejecute un *segue*, se avisará al método `prepareForSegue:sender:` del controlador donde se originó. Sobrescribiendo este método podremos saber cuándo se produce un determinado *segue*, y en ese caso realizar las acciones oportunas, como por ejemplo configurar los datos de la pantalla destino.

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
    [segue.destinationViewController setDetailItem:
        [NSString stringWithFormat:@"Item %d", indexPath.row]];
}
```

También podemos lanzar de forma programática un *segue* con el método `performSegueWithIdentifier:sender:`. Cada *segue* tendrá asociado un identificador, que se le asignará en la propiedad *Identifier* de la sección *Storyboard Segue* de su inspector de atributos.

En el caso de tener un *segue* de tipo Modal, podremos especificar en su propiedad *Transition* el estilo de la transición a la siguiente pantalla.



Atributos de los segues

Si queremos crear una transición personalizada, podemos crear una subclase de `UIStoryboardSegue`, establecer el tipo de *segue* a *Custom* (propiedad *Style*), y especificar la clase en la que hemos implementado nuestro propio tipo de *segue* en la

propiedad *Segue Class*.

11.4.2. Tablas en el storyboard

Con el uso de los *storyboards* la gestión de las celdas de las tablas se simplifica notablemente. Cuando introduzcamos una vista de tipo tabla, en su inspector de atributos (sección *Table View*), veremos una propiedad *Content* que puede tomar dos posibles valores:

- *Dynamic Prototypes*: Se utiliza para tablas dinámicas, en las que tenemos un número variable de filas basadas en una serie de prototipos. Podemos crear de forma visual uno o más prototipos para las celdas de la tabla, de forma que no será necesario configurar el aspecto de las celdas de forma programática, ni cargarlas manualmente.
- *Static Cells*: Se utiliza para tablas estáticas, en las que siempre se muestra el mismo número de filas. Por ejemplo, nos puede servir para crear una ficha en la que se muestren los datos de una persona (nombre, apellidos, dni, telefono y email).

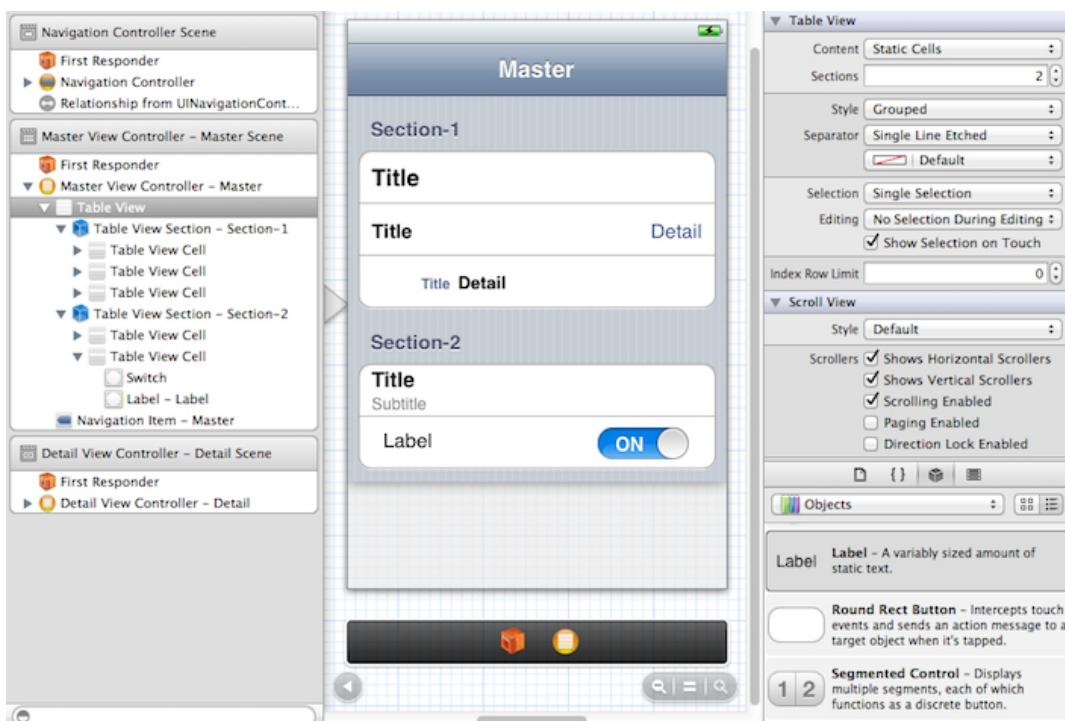


Tabla estática en el storyboard

En el caso de las tablas dinámicas, si definimos más de un prototipo, podemos dar a cada uno de ellos un identificador de reutilización diferente, para así en el código poder seleccionar el tipo deseado (esto lo especificaremos en el inspector de atributos de la celda prototípico, sección *Table View Cell*, propiedad *Identifier*).

Además, en el código se nos asegura que el método

`dequeueReusableCellWithIdentifier:`: siempre nos va a dar una instancia de una celda correspondiente al prototipo cuyo identificador se especifique como parámetro. Por lo tanto, ya no será necesario comprobar si nos ha devuelto `nil` y en tal caso instanciarla nosotros, como hacíamos anteriormente, sino que podremos confiar en que siempre nos devolverá una instancia válida.

```
UITableViewCell *cell = [tableView  
    dequeueReusableCellWithIdentifier:CellIdentifier];  
  
cell.textLabel.text = @"Item";
```

Las tablas estáticas podrán crearse íntegramente en el *storyboard*. Cuando creamos una tabla de este tipo podremos especificar el número de secciones (propiedad *Sections* en *Table View*), y en el *dock* aparecerá un nodo para cada sección, que nos permitirá cambiar sus propiedades. Dentro de cada sección podremos también configurar el número de filas que tiene (propiedad *Rows* en *Table View Section*), y veremos cada una de estas filas de forma visual en el editor, pudiendo editarlas directamente en este entorno.

Para cada celda podemos optar por utilizar un estilo predefinido, o bien personalizar su contenido (*Custom*). Esto lo podemos configurar en la propiedad *Style* de *Table View Cell*. En el caso de optar por celdas personalizadas, deberemos arrastrar sobre ellas las vistas que queramos que muestren.

También podremos conectar las celdas mediante *segues*, de forma que cuando se pulse sobre ellas se produzca una transición a otra pantalla.

12. Ejercicios de transiciones y storyboards

12.1. Pestañas

Cambiaremos el punto de entrada de la aplicación, para que ahora el controlador principal sea un controlador basado en pestañas. Tendremos tres pestañas: *Películas*, *Búsqueda* y *Configuración*. Se pide:

- a) Utilizaremos el controlador de navegación inicial (`UAMasterViewController`) como contenido para la pestaña *Películas*, y el controlador de búsqueda creado en la sesión anterior (`UABusquedaViewController`) para la pestaña *Búsqueda*. Vamos a añadir un controlador adicional `UAConfiguracionViewController`, que será de tipo `UIViewController`, para la pestaña *Configuración*.
- b) Vamos a crear un `UITabBarController` de forma programática en el *app delegate*. Añadiremos a este controlador las tres pestañas indicadas anteriormente, cada una de ellas instanciando el controlador correspondiente.

Ayuda

Esto deberá realizarse en el método `application:didFinishLaunchingWithOptions:` de `UAAppDelegate`. En lugar de poner como controlador raíz `UAMasterViewController`, crearemos un nuevo controlador de tipo `UITabBarController` que hará de controlador raíz, y contendrá los tres controladores anteriores como pestañas.

- c) Modificar el inicializador designado de cada controlador para incluir la información del título y el ícono de la pestaña. Puedes descargar las plantillas de la sesión para obtener los iconos necesarios e incluirlos en el proyecto (los íconos de las pestañas se encuentran en una carpeta `iconos_filmoteca`).

- d) De forma alternativa, podemos implementar todo lo anterior de forma visual en un fichero NIB. En este caso deberemos:

- Crear un NIB con el `UITabBarController` como elemento principal.
- Añadir tres controladores como pestañas, arrastrándolos desde la librería de objetos. Uno de ellos debe ser de tipo `UINavigationController` y los otros de tipo `UIViewController`.
- Modificar el atributo `Class` del inspector de identidad de los controladores `UIViewController` anteriores para establecer su tipo (`UABusquedaViewController` y a `UAConfiguracionViewController`).
- Establecer el tipo del controlador raíz del controlador de navegación como `UAMasterViewController`.
- Indicar para cada pestaña su título y su ícono en los campos del inspector de atributos.
- En el *app delegate* cargar el contenido del NIB anterior y asigna el

`UITabBarController` raíz a `window.rootViewController`. Crea el *outlet* necesario para tener acceso a dicho controlador.

Importante

Recuerda que en este caso los controladores que se cargan del NIB no se inicializarán mediante sus inicializadores, sino que se recuperará el objeto ya creado directamente del NIB. Por lo tanto, toda la inicialización necesaria debería realizarse en el método `awakeFromNib`, o en `viewDidLoad`, no en su inicializador designado.

12.2. Aplicación basada en storyboards

Vamos a crear una nueva aplicación utilizando storyboards. Se pide:

- a) Crear un nuevo proyecto basado en pestañas (*Tabbed Application*) llamado `Tareas`, que utilice *storyboards* y ARC.
- b) Crear 3 pestañas (*tabs*). La primera (`Tareas`) deberá contener un controlador de navegación, la segunda (`Configuración`) un controlador tipo tabla estática, y la tercera (`Acerca de`) un controlador básico.
- c) Asignar un ícono a cada pestaña. Para ello importaremos en el proyecto el conjunto de iconos que se encuentran en el directorio `iconos_tareas` de las plantillas de la sesión.
- d) En el controlador de navegación (`Tareas`), crear como pantalla raíz una tabla dinámica (lista de tareas), y como pantalla secundaria un controlador de tabla estática (datos de la tarea).
- e) En el controlador de la pestaña `Acerca de`, pon una imagen (por ejemplo el logo del curso) y el nombre del autor mediante una etiqueta de texto.
- f) En la tabla estática de la pantalla de configuración crear dos secciones, con 3 filas la primera de ellas, y 2 filas la segunda. En la primera sección tendremos:
 - Fila de tipo *custom*, con un `UITextField` donde introducir un mensaje, que tendrá como *placeholder* el texto *Ej. Mensaje de alerta*.
 - Fila de tipo *custom*, con un `UISwitch` que nos permite activar o desactivar el sonido de las alertas.
 - Fila de tipo *custom*, con un `UISwitch` que nos permite activar o desactivar la aparición alertas.

En la segunda sección, las filas serán:

- Fila de tipo *right detail* con la versión de la aplicación.
- Fila de tipo *right detail* con el número de *build* de la aplicación.

El aspecto de esta pantalla deberá ser el siguiente:



Pantalla de configuración

Sólo implementaremos el aspecto de esta interfaz. Por simplicidad no implementaremos las funcionalidades asociadas a estas opciones.

g) Dentro de las pantallas de la sección *Tareas*:

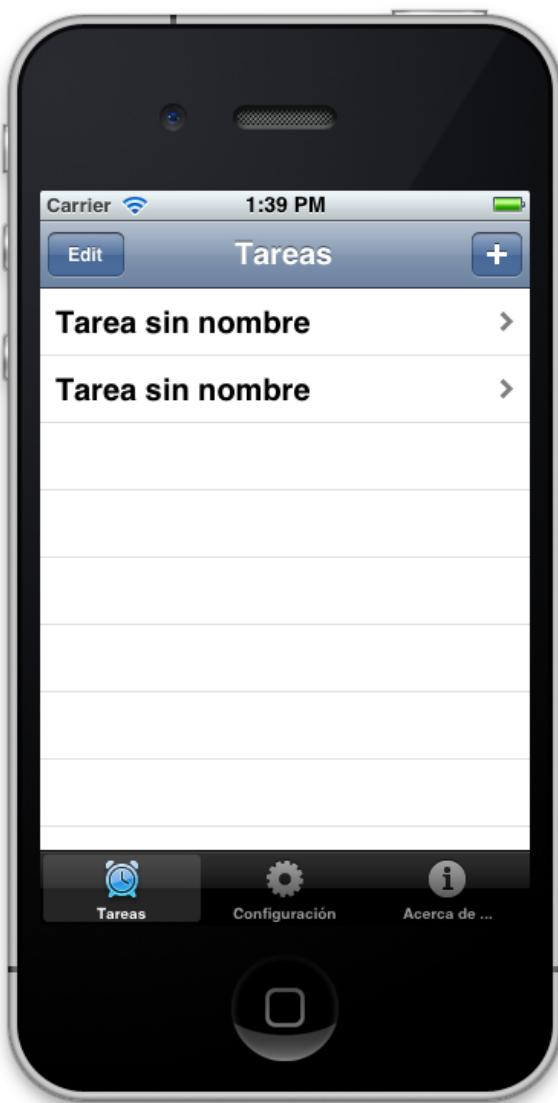
- En la pantalla principal (lista de tareas) crearemos una celda prototipo de tipo *Basic* en la tabla dinámica.
- La celda prototipo tendrá un identificador de reutilización `TareaCell`.
- Conectaremos la celda con la pantalla de datos de la tarea mediante un *segue*, que tendrá como identificador `TareaSeleccionada` y será de tipo *push*.
- La pantalla con la lista tendrá como título *Tareas* en la barra de navegación, y un

botón para añadir nuevos *items* a la lista (utilizar el botón del sistema de tipo *Add*, atributo *Identifier*).

- La pantalla con los datos de la tarea tendrá como título *Tarea* en la barra de navegación, y un botón para guardar los cambios realizados en la tarea.
- La tabla estática de la pantalla de datos tendrá una única fila, de tipo *custom*, que contendrá un campo de texto editable con el nombre de la tarea seleccionada. Tendrá como *placeholder* el texto *Ej. Entregar trabajo*, y durante la edición deberá aparecer el botón para vaciar el texto (*Clear Button*)

h) Crear un controlador para la pantalla con la lista de tareas, al que llamaremos `UATareasViewController`. Será de tipo `UITableViewController`, y lo asociaremos a la pantalla de la lista de tareas en el *storyboard* mediante el atributo *Class* del inspector de identidades. En dicho controlador implementaremos lo siguiente:

- Crearemos un `NSMutableArray` como propiedad de la clase, que contendrá la lista de tareas almacenadas. Para cada tarea almacenaremos en la lista únicamente un `NSString` con su nombre (no es necesario crear una nueva clase para las tareas).
- Implementar los métodos de *data source* (`numberOfSectionsInTableView:`, `tableView:numberOfRowsInSection:`, y `tableView:cellForRowAtIndexPath:`) para poblar la tabla de datos a partir de la lista anterior.
- Añadiremos una acción para el botón de la barra de navegación para agregar una nueva tarea. En dicha acción insertaremos una nueva tarea de nombre `@"Nueva tarea"`, y la insertaremos en la última posición de la tabla mediante una animación.
- Añadiremos el botón de edición proporcionado por el sistema como botón izquierdo de la barra de tareas. Esto lo haremos de forma programática en el método `viewDidLoad` del controlador.
- Implementaremos el método `tableView:commitEditingStyle:forRowAtIndexPath:` del delegado para permitir el borrado de tareas.
- Implementaremos el método `tableView:moveRowAtIndexPath:toIndexPath:` del delegado para permitir la reordenación de tareas.



Pantalla de lista de tareas

i) Crear un controlador para la pantalla con los datos de una tarea, al que llamaremos `UATareaViewController`, también de tipo `UITableViewController`. En él implementaremos lo siguiente:

- Una propiedad `tarea` de tipo `NSString` que recogerá el nombre de la tarea que vamos a editar.
- Una propiedad `indexPath` de tipo `NSIndexPath` que almacenará la posición de la tarea que estamos editando.
- Una acción `guardarTarea`, vinculada al botón *Guardar* de la barra de navegación (implementaremos su código en el siguiente apartado)
- Adoptaremos el protocolo `UITextFieldDelegate`, e implementaremos el método

`textFieldShouldReturn:`, en el que llamaremos a `guardarTarea` y devolveremos YES para que se cierre el teclado tras editar.

- Al tratarse de una tabla estática, no implementaremos ningún método de *data source* (y tampoco del delegado de la tabla).
- Crearemos un *outlet* `textFieldTarea`, conectado con el campo de texto que tenemos en la tabla estática.
- En `viewDidLoad` mostraremos en el campo de texto anterior el texto que hayamos recibido en la propiedad `tarea`.



Pantalla de datos de la tarea

j) Establecer la comunicación entre la lista de tareas y los datos de la tarea. Para ello

deberemos:

- Para que la comunicación sea bidireccional (es decir, que la lista de tareas puede recibir el aviso de que una tarea ha sido editada), vamos a seguir el patrón delegado. Crearemos en `UATareaViewController.h` un protocolo para el delegado de la pantalla de edición de tareas:

```
@class UATareaViewController;

@protocol UATareaViewControllerDelegate <NSObject>
- (void)tareaViewController:(UATareaViewController *)controller
didChangeTarea:(NSString *)tarea
atIndexPath:(NSIndexPath *)indexPath;
@end
```

- Creamos una propiedad `delegate` en la clase `UATareaViewController`:

```
@property (nonatomic, unsafe_unretained)
id<UATareaViewControllerDelegate> delegate;
```

- Cuando pulsemos sobre el botón para guardar una tarea, cogeremos el valor introducido en el campo de texto y se lo notificaremos al delegado. Tras esto, volveremos a la pantalla anterior:

```
[self.delegate tareaViewController:self
didChangeTarea:self.textFieldTarea.text
atIndexPath:selfIndexPath];
[self.navigationController popViewControllerAnimated:YES];
```

- Hacer que `UATareasViewController` adopte el protocolo anterior, implementando el método correspondiente:

```
- (void)tareaViewController:(UATareaViewController *)controller
didChangeTarea:(NSString *)tarea
atIndexPath:(NSIndexPath *)indexPath {
[self.tareas replaceObjectAtIndex:indexPath.row withObject:tarea];
[self.tableView
reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];
}
```

- Implementar en `UATareasViewController` el método `prepareForSegue:sender:` para configurar el controlador destino en el momento en el que se realiza la transición "TareaSeleccionada":

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
if([segue.identifier isEqualToString:@"TareaSeleccionada"]) {
UATareaViewController *controller =
segue.destinationViewController;
controller.delegate = self;
controllerIndexPath = self.tableView.indexPathForSelectedRow;
controller.tarea = [self.tareas objectAtIndex:
self.tableViewIndexPathForSelectedRow.row];
}
}
```

- k) Guarda la lista de tareas en disco cuando la aplicación se mueva a segundo plano. En este caso, ¿podemos utilizar directamente los métodos que nos proporciona la clase

NSArray? Utiliza el método más sencillo que puedas para realizar el almacenamiento. En el inicializador de `UATareasViewController` se deberá intentar cargar la lista de tareas del disco.

Ayuda

Al trabajar con *storyboards* no tenemos una relación directa entre la clase `UAAppDelegate` (que recibe el evento que nos notifica que la aplicación se ha movido a segundo plano), y la clase `UATareasViewController` (que es la que contiene la lista de tareas, y por lo tanto la encargada de guardarla en disco). Sin embargo, hemos visto que existe una forma para comunicar clases que no están directamente relacionadas en el diagrama de clases. Busca entre la documentación `UIApplicationDidEnterBackgroundNotification`.

13. Componentes para iPad y aplicaciones universales

En este módulo comentaremos distintos componentes de diseño de interfaz de usuario para sistemas iOS. Comenzaremos explicando el funcionamiento de los controladores específicos de iPad para continuar detallando la manera de cómo podemos programar aplicaciones universales (compatibles tanto para iPhone como para iPad). En la segunda sesión nos centraremos en comentar las distintas guías de estilo que recomienda Apple para el diseño de aplicaciones iOS, algo que deberemos cumplir en la medida de lo posible en el proceso de diseño de la interfaz de usuario.

13.1. Componentes específicos para iPad

Debido a las diferencias evidentes de tamaño entre un iPhone y un iPad, *Apple* ha desarrollado una API específica para este último, de esta forma se pretende que aquellas aplicaciones que estén disponibles en iPad utilicen esta API en la medida de lo posible ya que así se mejorará considerablemente la experiencia de usuario y la usabilidad.

Con la salida del iPad al mercado aparecieron un par de nuevos controladores enfocados exclusivamente a este. El uso de ambos está unido muy a menudo debido a que se complementan mutuamente, estos son los controladores:

- **Split View:** Formado por dos vistas independientes, una situada en la parte izquierda y la otra en la parte derecha. Divide la pantalla principal en dos y se utiliza muy frecuentemente para mostrar en la vista de la izquierda el listado de opciones de un menú en forma de tabla y en la parte de la derecha el detalle de la vista seleccionada.
- **Popover:** Es una especie de ventana emergente que puede aparecer en cualquier parte de la pantalla, normalmente junto a un botón al pulsarlo. Muy útil para mostrar un listado de opciones o una tabla con información dentro de un *Split View*. *Apple* recomienda usar un popover para mostrar el listado de opciones de la tabla de la parte de la izquierda de un *Split View* cuando este está en posición vertical (*portrait*).

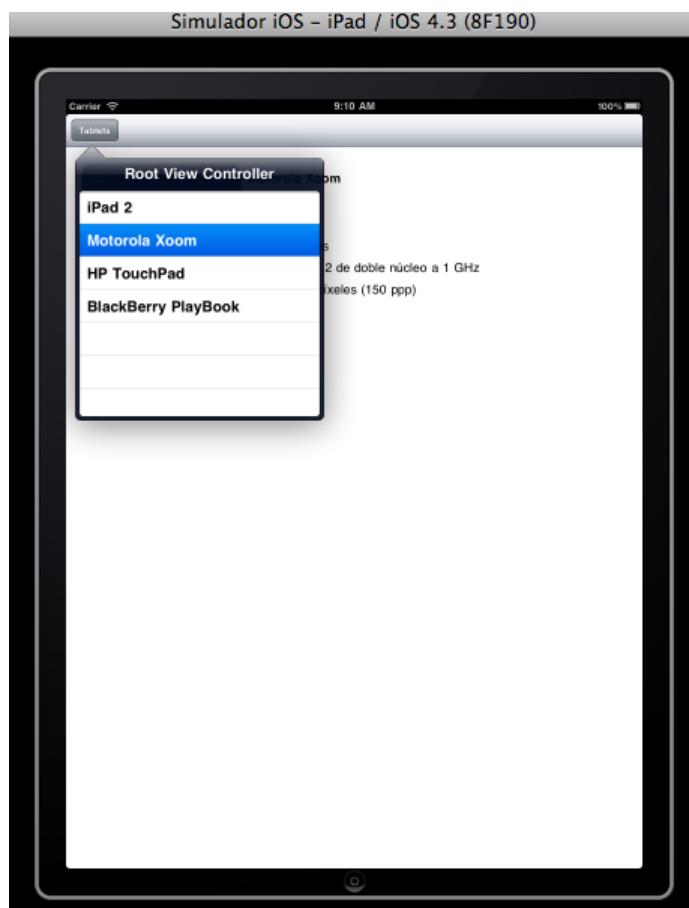
13.1.1. Split View

Un **Split View** o *Vista dividida* es, como hemos comentado anteriormente, una combinación de dos vistas, la primera de las cuales es equivalente a una vista "tipo" de iPhone cuando este se encuentra en orientación vertical, usando incluso la misma anchura. Esta vista se utilizará principalmente para la navegación principal dentro de la aplicación. Por otro lado en la segunda vista, que corresponderá a la porción más grande de la pantalla, mostraremos la información que hayamos seleccionado desde la vista de la izquierda en detalle. Comentar que este es el uso que *Apple* propone para este tipo de controlador aunque puede tener muchos más.



Split View

En modo vertical (*Portrait Orientation*), el *Split View* cambia y la vista de la izquierda (el menú) pasa ahora a ser una vista de tipo *popover* que aparecerá, cuando se pulsa un botón a modo de cuadro de diálogo desde la parte superior izquierda de la pantalla. Esta transformación hace falta programarla, aunque es bastante sencillo. De esta forma quedaría la vista de la derecha ocupando toda la pantalla del iPad.



Split View vertical

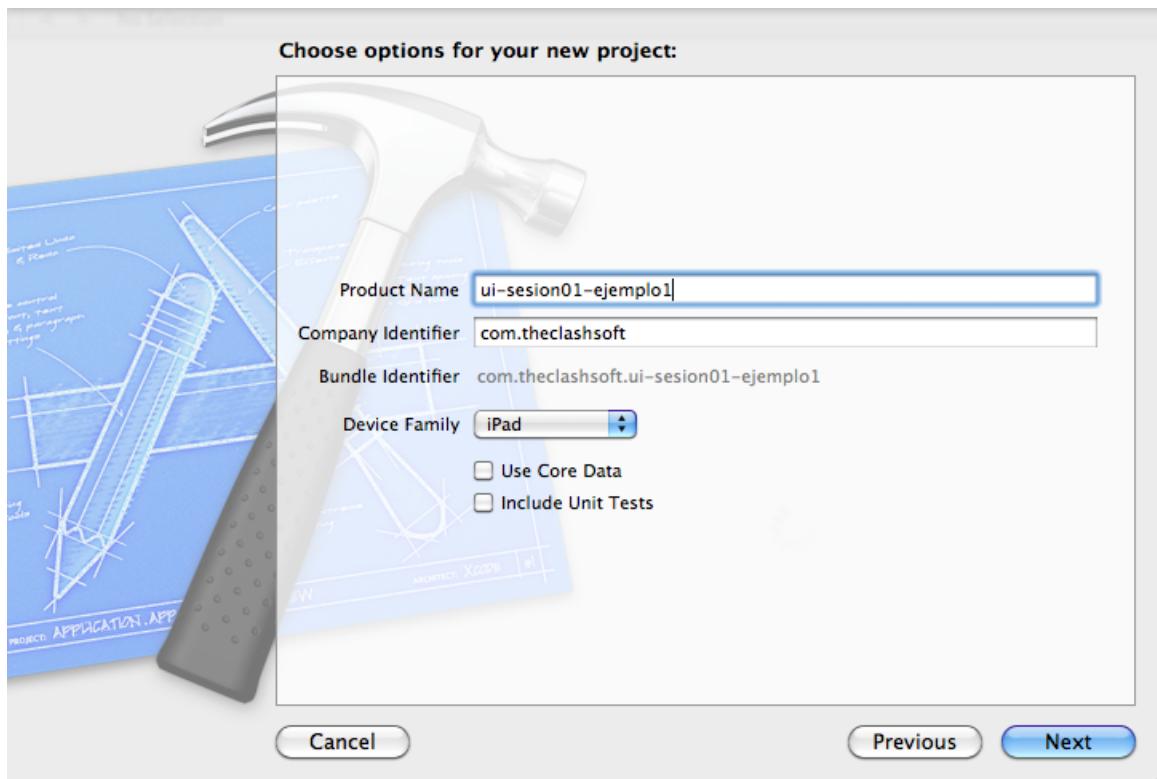
Incorporar un controlador *Split View* en nuestra aplicación es bastante sencillo aunque puede existir alguna confusión a la hora de implementarlo, es por ello que vamos a explicarlo con un sencillo ejemplo el cual lo usaremos más adelante como punto de partida para nuestras aplicaciones. En la aplicación de ejemplo realizaremos una comparativa simple de los distintos tablets que existen en el mercado: tendremos un listado en forma de tabla de todos los tablets en la parte izquierda del *Split View* y cuando seleccionemos uno nos aparecerán sus detalles en la parte derecha. Para finalizar el ejemplo, cuando tengamos todo funcionando, implementaremos un *popover* para que cuando rotemos el iPad a posición vertical, podamos navegar por las distintas opciones.

13.1.1.1. Creando el proyecto

A pesar de que XCode nos da la opción de crear un proyecto de tipo `UISplitView` en el que tendremos un controlador *Split View* desde el inicio, nosotros vamos a crearlo desde cero, ya que para futuras aplicaciones necesitaremos saber cómo incorporar un *Split View* en cualquier vista.

Plantilla Master Detail Application

A partir de la versión 4.2 de XCode y coincidiendo con la aparición de iOS 5, XCode renovó todas las plantillas que disponía para crear nuevos proyectos iOS, entre las que añadió se encuentra una que se llama *Master Detail Application*. Al seleccionar esta plantilla a la hora de crear un nuevo proyecto, XCode nos creará la estructura básica para una aplicación Universal (por defecto) usando la controladora *UISplitViewController* como base.



Crea proyecto

Por lo tanto, empezamos creando un proyecto usando la plantilla *Single View Application*, seleccionamos en la lista de la familia de dispositivos *iPad*, deseleccionamos *Use Storyboard* y *Use Automatic Reference Counting* y hacemos click en *Next*. Guardamos el proyecto con el nombre *ui-sesion01-ejemplo1*. De esta forma tenemos ya la estructura básica de un proyecto diseñado para iPad, si lo ejecutamos nos aparecerá el simulador de iPad con la pantalla en blanco.

13.1.1.2. Creando las clases de las vistas

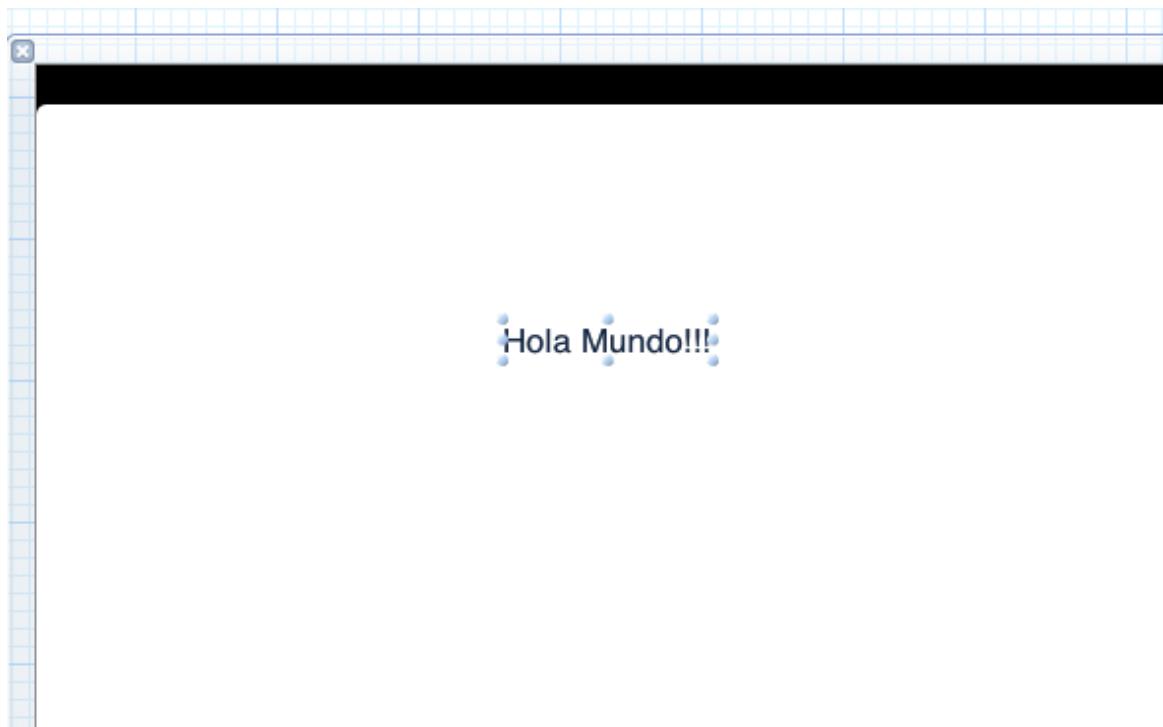
Ahora vamos a crear las dos vistas del *Split View* como nosotros queremos: por un lado crearemos una vista de tipo *Table View Controller* que será la de la parte izquierda y por otro lado una de tipo *View Controller* para la parte derecha del *Split View*.

Comenzamos creando el Table View Controller, para ello hacemos click en *File > New File* y seleccionamos *Cocoa Touch > UIViewController Subclass* y seleccionamos en el siguiente paso "Subclass off" *UITableViewController* y nos aseguramos de tener marcada la opción de *Targeted for iPad*. Lo guardamos con el nombre de *MenuViewController*.

Ahora abrimos el fichero *MenuViewController.m* y modificamos algún dato de la clase: en el método *numberOfSectionsInTableView* devolvemos 1 y en el método *numberOfRowsInSection* devolvemos 5. De esta forma tendremos una tabla con cinco filas vacías, esto lo hacemos para hacer una prueba rápida, más adelante meteremos datos útiles.

Ahora pasamos a crear la vista vacía *UIViewController*, para ello hacemos click en *New > New File* y seleccionamos *UIViewController Subclass*. Después seleccionamos "Subclass of" *UIViewController* y nos aseguramos de marcar la opción de *Targeted for iPad*. Guardamos con el nombre de *DetalleViewController* y esta vista será la que corresponda a la parte derecha del *Split View*.

Para hacer una prueba rápida, vamos a poner algo dentro de la vista de detalle, para ello abrimos la vista *DetalleViewController.xib*, arrastramos un *label* y escribimos por ejemplo *Hola Mundo!*. Con esto ya tenemos las dos clases creadas y listas para asignarlas a nuestro *Split View*.



Vista Split

13.1.1.3. Asignando las vistas

Una vez que tenemos creadas la vista *Master* y la vista de detalle vamos a crear un objeto de tipo `UISplitViewController` y a asignarle ambas vistas. Esto lo haremos dentro de la clase `didFinishLaunchingWithOptions` del fichero `ui_sesion01_ejemplolAppDelegate.m`:

`ui_sesion01_ejemplolAppDelegate.m`

```
#import "MenuViewController.h"
#import "DetalleViewController.h"

@synthesize window=_window;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:
NSDictionary *launchOptions
{
    self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]] autorelease];
    // Override point for customization after application launch.

    MenuViewController *listado = [[MenuViewController alloc]
initWithNibName:@"MenuViewController" bundle:nil];
    DetalleViewController *detalle = [[DetalleViewController alloc]
initWithNibName:@"DetalleViewController" bundle:nil];

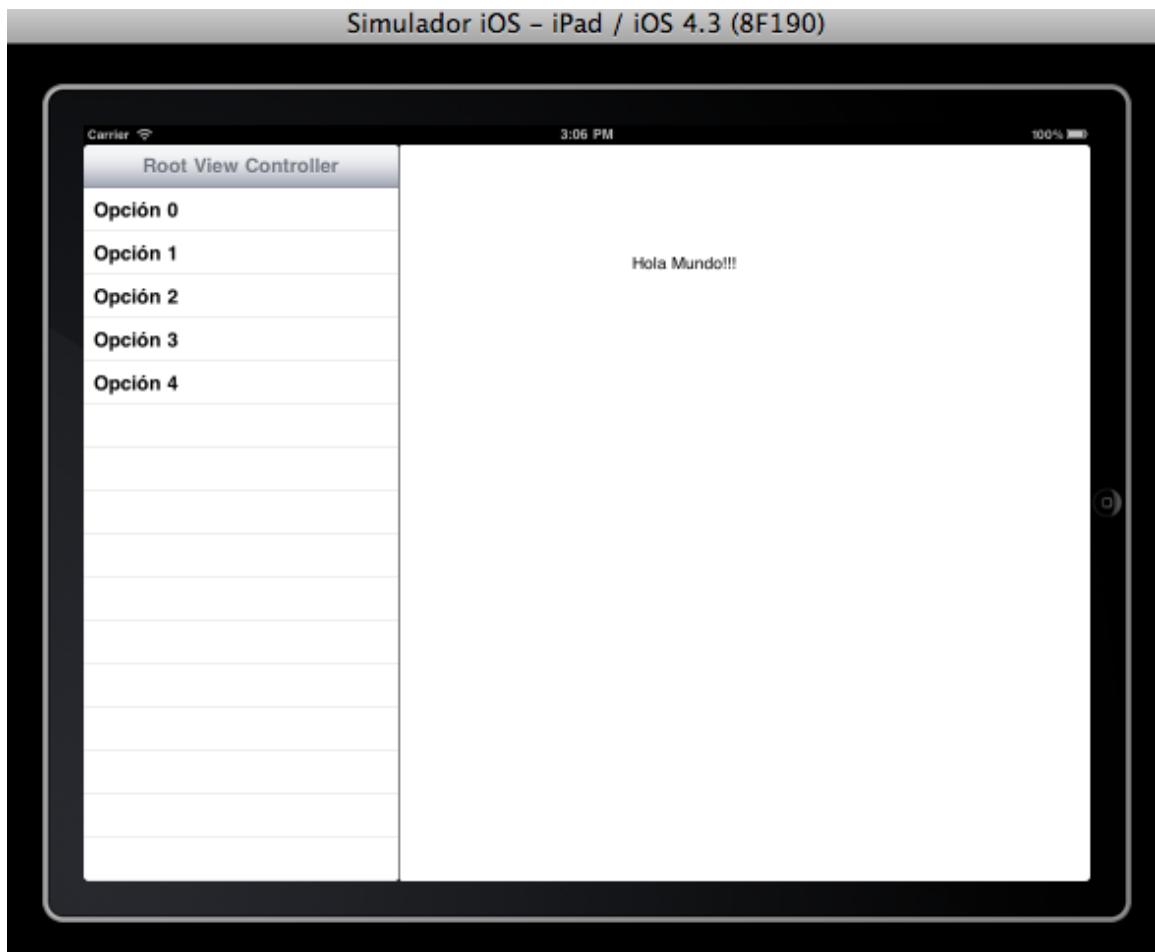
    UISplitViewController *splitViewController = [[UISplitViewController
alloc] init];
    [splitViewController setViewControllers:[NSArray
arrayWithObjects:listado,detalle, nil]];

    self.window.rootViewController = splitViewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

Nota

La programación de un *Split View Controller* se puede realizar de muchas formas. Esta que hemos explicado es la más sencilla y más clara. Otra manera de crearlo sería mediante las vistas. Añadiendo a la vista principal de la aplicación un objeto de tipo *Split View Controller* desde el Interface Builder.

Ahora ya podemos compilar y ejecutar el proyecto y ver los resultados. Si giramos el simulador podremos observar como se adapta el *Split View* de forma automática.



Split View iPad

13.1.1.4. Programando el modelo de datos

El siguiente paso es programar nuestro modelo de datos. Esto se podría hacer utilizando algún método de persistencia como *ficheros*, *SQLite* o *Core Data*, pero por simplificar no utilizaremos ninguno y simplemente crearemos los objetos en memoria directamente.

Vamos a programar la clase que representará cada tablet que queremos mostrar, para ello hacemos click en *New > New File* y seleccionamos *Objective-C class*, subclass of *NSObject*. Guardamos el fichero con el nombre *Tablet*. Ahora abrimos cada uno de los archivos creados y escribimos el siguiente código:

Tablet.h

```
#import <Foundation/Foundation.h>

@interface Tablet : NSObject {
```

```

NSString * _nombre;
NSString * _procesador;
NSString * _grafica;
NSString * _memoria;
NSString * _pantalla;
NSString * _resolucion;
NSString * _imagen;
}

@property (nonatomic, copy) NSString *nombre;
@property (nonatomic, copy) NSString *procesador;
@property (nonatomic, copy) NSString *grafica;
@property (nonatomic, copy) NSString *memoria;
@property (nonatomic, copy) NSString *pantalla;
@property (nonatomic, copy) NSString *resolucion;
@property (nonatomic, copy) NSString *imagen;

- (Tablet *)initWithNombre:(NSString *)nombre
    procesador:(NSString *)procesador
        grafica:(NSString *)grafica
        memoria:(NSString *)memoria
        pantalla:(NSString *)pantalla
        resolucion:(NSString *)resolucion
            imagen:(NSString *)imagen;

@end

```

Tablet.m

```

#import "Tablet.h"

@implementation Tablet

@synthesize nombre = _nombre;
@synthesize procesador = _procesador;
@synthesize grafica = _grafica;
@synthesize memoria = _memoria;
@synthesize pantalla = _pantalla;
@synthesize resolucion = _resolucion;
@synthesize imagen = _imagen;

- (Tablet *)initWithNombre:(NSString *)nombre
    procesador:(NSString *)procesador
        grafica:(NSString *)grafica
        memoria:(NSString *)memoria
        pantalla:(NSString *)pantalla
        resolucion:(NSString *)resolucion
            imagen:(NSString *)imagen
{
    if ((self = [super init])) {
        self.nombre = nombre;
        self.procesador = procesador;
        self.grafica = grafica;
        self.memoria = memoria;
        self.pantalla = pantalla;
        self.resolucion = resolucion;
        self.imagen = imagen;
    }
    return self;
}

```

```
}

-(void) dealloc {
    self.nombre = nil;
    self.procesador = nil;
    self.grafica = nil;
    self.memoria = nil;
    self.pantalla = nil;
    self.resolucion = nil;
    self.imagen = nil;

    [super dealloc];
}

@end
```

Con este código ya tenemos el modelo de datos definido. Ahora vamos a implementar la parte izquierda del *Split View*.

13.1.1.5. Programando la vista detalle

La parte izquierda del *Split View* es una tabla en donde aparecerán los nombres de los distintos modelos de tablets, para implementar esto añadiremos una nueva variable a la clase `MenuViewController` que será un array con los objetos `Tablet`. Abrimos `MenuViewController.h` y escribimos lo siguiente:

```
#import <UIKit/UIKit.h>

@interface MenuViewController : UITableViewController {
    NSMutableArray *_tablets;
}

@property (nonatomic, retain) NSMutableArray *tablets;
@end
```

Ahora añadimos las siguientes líneas en el archivo `MenuViewController.m`:

```
// Justo debajo del #import
#import "Tablet.h"

// Debajo de @implementation
@synthesize tablets = _tablets;

// En numberOfRowsInSection, cambia el 5 con lo siguiente:
return [_tablets count];

// En cellForRowAtIndexPath, después de "Configure the cell..."
Tablet *tablet = [_tablets objectAtIndex:indexPath.row];
cell.textLabel.text = tablet.name;

// En dealloc
self.tablets = nil;
```

Con esto ya tenemos el *Table View* terminado. Ahora vamos a crear los objetos tablets en la clase delegada, para ello abrimos `uisession01_ejemplo1AppDelegate.m` y escribimos el siguiente código dentro del método `didFinishLaunchingWithOptions` justo antes de `[window addSubview:_splitViewController.view];`

```

Tablet *ipad2 = [[Tablet alloc] initWithNombre:@"iPad 2"
                                         procesador:@"Apple A5 de doble
                                         núcleo a 1 GHz"
                                         grafica:@"PowerVR SGX543MP2"
                                         memoria:@"512 MB"
                                         pantalla:@"9,7 pulgadas"
                                         resolucion:@"1024 x 768 píxeles
                                         (132 ppp)"
                                         imagen:@"/ipad2.jpg"];

Tablet *motorola = [[Tablet alloc] initWithNombre:@"Motorola Xoom"
                                         procesador:@"Nvidia Tegra 2 de
                                         doble núcleo a
                                         1 GHz"
                                         grafica:@"GeForce GPU 333"
                                         memoria:@"1 GB"
                                         pantalla:@"10,1 pulgadas"
                                         resolucion:@"1280 x 800 píxeles
                                         (150 ppp)"
                                         imagen:@"/motorola.jpg"];

Tablet *hp = [[Tablet alloc] initWithNombre:@"HP TouchPad"
                                         procesador:@"Qualcomm Snapdragon
                                         de doble núcleo
                                         a 1,2 GHz"
                                         grafica:@"Adreno 220"
                                         memoria:@"1 GB"
                                         pantalla:@"9,7 pulgadas"
                                         resolucion:@"1024 x 768 píxeles
                                         (132 ppp)"
                                         imagen:@"/hp.jpg"];

Tablet *blackBerry = [[Tablet alloc] initWithNombre:@"BlackBerry
                                         PlayBook"
                                         procesador:@"Procesador de doble
                                         núcleo a 1 GHz"
                                         grafica:@"Desconocido"
                                         memoria:@"1 GB"
                                         pantalla:@"7 pulgadas"
                                         resolucion:@"1024 x 600 píxeles"
                                         imagen:@"/blackberry.jpg"];

NSMutableArray *listaTablets = [NSMutableArray array];
[listaTablets addObject:ipad2];
[listaTablets addObject:motorola];
[listaTablets addObject:hp];
[listaTablets addObject:blackBerry];

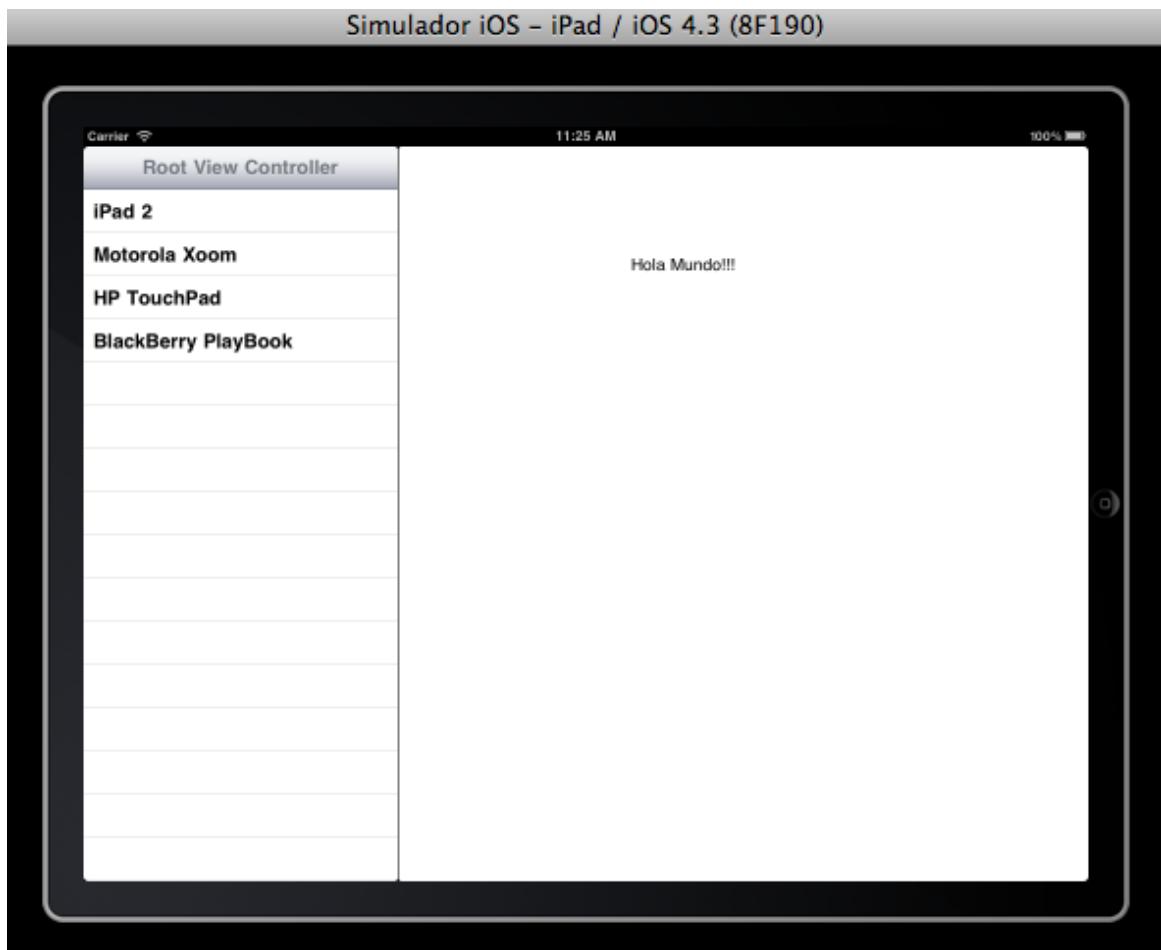
_menuViewController.tablets = listaTablets;

```

Imágenes

Las imágenes de las tablets las podemos descargar desde [aquí](#). Una vez descargadas, las descomprimimos y tenemos arrastrarlas al directorio de Supporting Files.

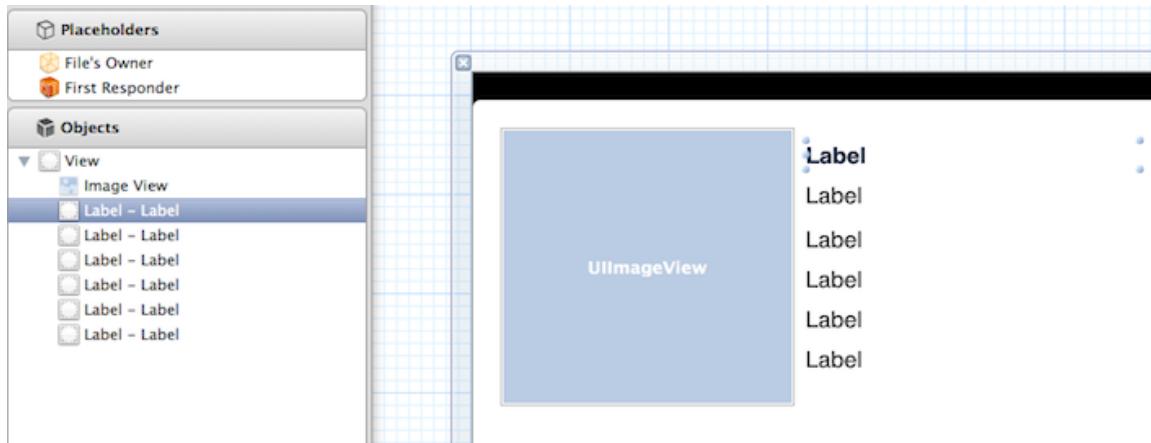
Compilamos ahora y si todo ha ido bien nos debe de aparecer lo siguiente:



Split View iPad 2

13.1.1.6. Programando la vista derecha o principal

Una vez programado el menu del *Split View*, vamos a diseñar y programar la parte derecha, o sea, la vista de detalle. Abrimos el fichero `DetalleViewController.xib`, borramos el `Label` creado anteriormente y diseñamos la vista de la manera que se muestra en la siguiente imagen:



Estructura vista detalle Split View

Como podemos ver en la imagen anterior, hemos añadido los siguientes componentes a la vista de detalle:

- UIImageView > imgTablet
- UILabel > labelNombre
- UILabel > labelProcesador
- UILabel > labelGrafica
- UILabel > labelMemoria
- UILabel > labelPantalla
- UILabel > labelResolucion

Ahora abrimos el fichero `DetalleViewController.h` y añadimos los "Outlets" para poder referenciarlos desde la vista que hemos creado:

```
#import <UIKit/UIKit.h>
#import "Tablet.h"

@interface DetalleViewController : UIViewController {
    Tablet *_tablet;
    UIImageView *_imgTablet;
    UILabel *_labelNombre;
    UILabel *_labelProcesador;
    UILabel *_labelGrafica;
    UILabel *_labelMemoria;
    UILabel *_labelPantalla;
    UILabel *_labelResolucion;
}

@property (nonatomic, retain) Tablet *tablet;
@property (nonatomic, retain) IBOutlet UIImageView *imgTablet;
@property (nonatomic, retain) IBOutlet UILabel *labelNombre;
@property (nonatomic, retain) IBOutlet UILabel *labelProcesador;
@property (nonatomic, retain) IBOutlet UILabel *labelGrafica;
@property (nonatomic, retain) IBOutlet UILabel *labelMemoria;
@property (nonatomic, retain) IBOutlet UILabel *labelPantalla;
@property (nonatomic, retain) IBOutlet UILabel *labelResolucion;
```

```
@end
```

En `DetalleViewController.m` escribimos lo siguiente debajo de `@implementation DetalleViewController`

```
    @synthesize tablet = _tablet;
    @synthesize imgTablet = _imgTablet;
    @synthesize labelNombre = _labelNombre;
    @synthesize labelGrafica = _labelGrafica;
    @synthesize labelMemoria = _labelMemoria;
    @synthesize labelPantalla = _labelPantalla;
    @synthesize labelProcesador = _labelProcesador;
    @synthesize labelResolucion = _labelResolucion;
```

Ahora abrimos la vista `DetalleViewController.xib` y enlazamos todos los "Outlets" a su objeto correspondiente. Con esto ya tenemos la vista de detalle creada, ahora sólo nos falta enlazarla con la vista de la izquierda, que es lo que vamos a hacer a continuación.

13.1.1.7. Enlazando las dos vistas

Existen muchas formas de programar la correspondencia entre la vista de la izquierda con la de la derecha, una de ellas es la que se propone en la plantilla de *Split View* de XCode, en el que se incluye la vista de detalle en la vista del menu, y desde esta se accede a las propiedades de la vista detalle. Este método no es muy recomendado usarlo ya que dificulta enormemente la reutilización de las vistas en otros proyectos o en este mismo.

Nosotros vamos a utilizar el método recomendado por *Apple* que se explica en [este link](#), usando un objeto delegado que recibe una petición desde la clase principal cuando se selecciona una fila en la tabla del menu de la izquierda.

La idea principal es que vamos a definir un protocolo con un sólo método que llamaremos `selectedTabletChanged`. El lado derecho se encargará de implementar dicho método y el lado izquierdo será el que realice la llamada mediante una clase delegada.

Para implementar esto debemos de crear una nueva clase *New > New File* seleccionando *Subclass of NSObject*. Guardamos la clase con el nombre `TabletSelectionDelegate`. Borramos el fichero `.m` ya que no lo necesitaremos y escribimos lo siguiente en `TabletSelectionDelegate.m`:

```
#import <Foundation/Foundation.h>

@class Tablet;

@protocol TabletSelectionDelegate
- (void)tabletSelectionChanged:(Tablet *)curSelection;
@end
```

Ahora modificamos el archivo `MenuViewController.h` para incluir `TabletSelectionDelegate`:

```
#import <UIKit/UIKit.h>
#import "DetalleViewController.h"
#import "TabletSelectionDelegate.h"

@interface MenuViewController : UITableViewController {
    NSMutableArray *_tablets;
    DetalleViewController *_detalleViewController;
    id<TabletSelectionDelegate> *_delegate;
}

@property (nonatomic, retain) NSMutableArray *tablets;
@property (nonatomic, retain) DetalleViewController
*detalleViewController;
@property (nonatomic, assign) id<TabletSelectionDelegate> *delegate;

@end
```

Y ahora cambiamos también `MenuViewController.m`:

```
// Debajo de @implementation
@synthesize delegate = _delegate;

// Dentro del metodo didSelectRowAtIndexPath:
if (_delegate != nil) {
    Tablet *tablet = (Tablet *) [_tablets
objectAtIndex:indexPath.row];
    [_delegate tabletSelectionChanged:tablet];
}

// Dentro de dealloc
self.delegate = nil;
```

Ya queda menos, ahora modificaremos la clase del detalle `DetalleViewController.h` añadiendo el protocolo:

```
#import "TabletSelectionDelegate.h"

@interface DetalleViewController : UIViewController
<TabletSelectionDelegate> {
```

Editamos el método `viewDidLoad` en el que asignaremos los textos de las etiquetas y la imagen y añadimos la implementación del método del protocolo dentro de `DetalleViewController.m`:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib.

    if (self.tablet != nil){
```

```
    self.labelNombre.text = self.tablet.nombre;
    self.labelGrafica.text = self.tablet.grafica;
    self.labelMemoria.text = self.tablet.memoria;
    self.labelPantalla.text = self.tablet.pantalla;
    self.labelProcesador.text = self.tablet.procesador;
    self.labelResolucion.text = self.tablet.resolucion;

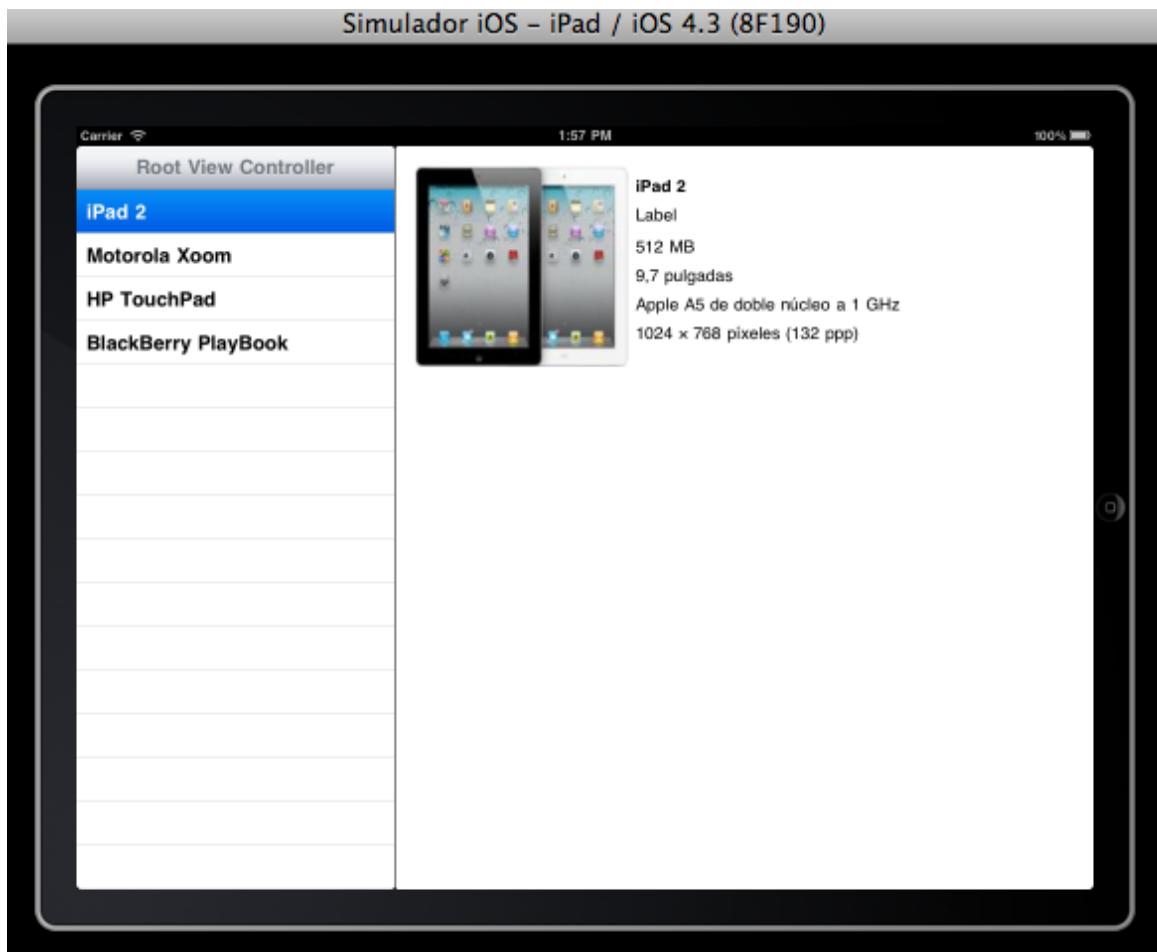
    self.imgTablet.image = [UIImage imageNamed:self.tablet.imagen];
}

- (void)tabletSelectionChanged:(Tablet *)curSelection {
self.tablet = curSelection;
[self viewDidLoad];
}
```

Finalmente tenemos que asignar el delegate que acabamos de crear dentro del método didFinishLaunchingWithOptions la clase delegada uisession01_ejemplo1AppDelegate.m:

```
_menuViewController.delegate = _detalleViewController;
```

Si está todo correcto, al compilar y ejecutar el proyecto debe de funcionar perfectamente el *Split View*:



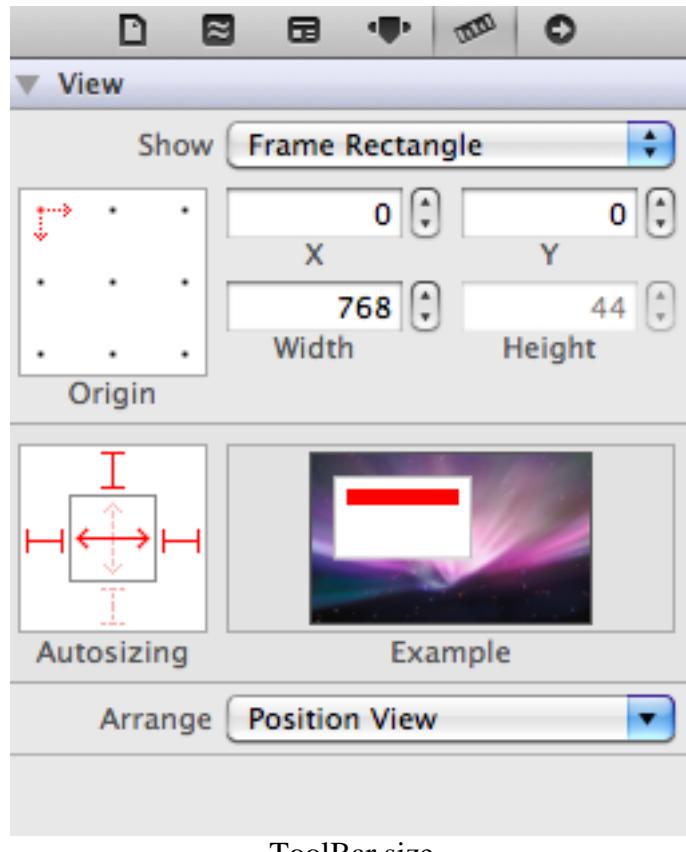
Split View terminado

Si giramos ahora el simulador (o el iPad) veremos que desaparece el menu de la izquierda y no podemos acceder a el, para evitar esto debemos de hacer uso de un elemento nuevo y exclusivo del iPad: *popover*. Aunque hablaremos con más detalle de el en el siguiente apartado, ahora debemos utilizarlo para mostrar una ventana emergente que contenga el menu de la izquierda. La programación de esta parte es muy sencilla y siempre es la misma, ¡vamos a implementarlo!

13.1.1.8. Añadiendo un Popover al Split View

La solución que propone *Apple* al problema comentando anteriormente es añadir un *Toolbar* con un *Button* en la parte derecha del *Split View* y al pulsar el botón que aparezca un *Popover* con el menu correspondiente a la parte izquierda. Para implementar esto debemos primero añadir el *Toolbar* dentro de la vista de la derecha, para ello abrimos *DetalleViewController.xib* y arrastramos el componente *UIToolbar* a la vista. Para que aparezca en la parte superior de la vista debemos de configurarlo de la siguiente

forma:



ToolBar size

Ahora hacemos los siguientes cambios a *DetalleViewController.h*:

```
// Añadimos UISplitViewControllerDelegate a la lista de protocolos
// de la interfaz
@interface DetalleViewController : UIViewController
<TabletSelectionDelegate,
UISplitViewControllerDelegate> {

// Dentro de la definición de la clase
UIPopoverController *_popover;
UIToolbar *_toolbar;

// Añadimos estas dos propiedades nuevas
@property (nonatomic, retain) UIPopoverController *popover;
@property (nonatomic, retain) IBOutlet UIToolbar *toolbar;
```

También debemos de añadir los siguientes métodos a *DetalleViewController.m*:

```
- (void)splitViewController:(UISplitViewController*)svc
willHideViewController:(UIViewController *)aViewController
withBarButtonItem:(UIBarButtonItem*)barButtonItem
forPopoverController:(UIPopoverController*)pc
```

```
{
    barButtonItem.title = @"Tablets";
    NSMutableArray *items = [[_toolbar items] mutableCopy];
    [items insertObject:barButtonItem atIndex:0];
    [_toolbar setItems:items animated:YES];
    [items release];
    self.popover = pc;
}

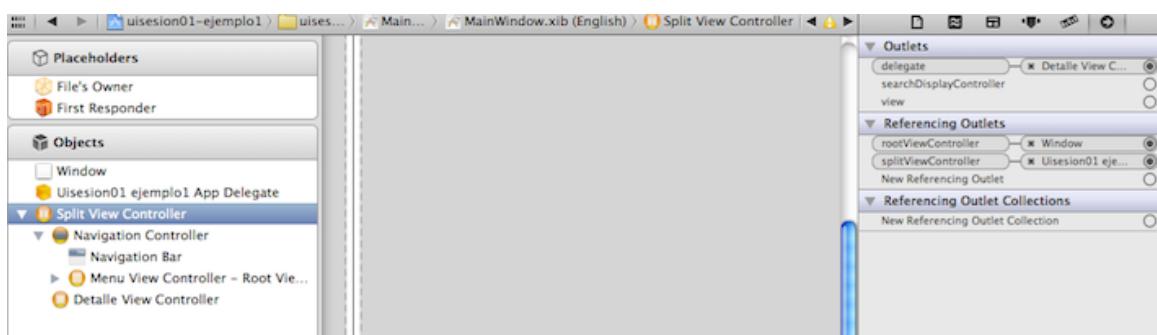
- (void)splitViewController: (UISplitViewController*)svc
    willShowViewController:(UIViewController *)aViewController
    invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
{
    NSMutableArray *items = [[_toolbar items] mutableCopy];
    [items removeObjectAtIndex:0];
    [_toolbar setItems:items animated:YES];
    [items release];
    self.popover = nil;
}
```

```
// Después de @implementation
@synthesize popover = _popover;
@synthesize toolbar = _toolbar;

// En dealloc
self.popover = nil;
self.toolbar = nil;

// En tabletSelectionChanged
if (_popover != nil) {
    [_popover dismissPopoverAnimated:YES];
}
```

Volvemos a abrir la vista `DetalleViewController.xib` y asignamos el "Outlet" al `ToolBar` que hemos añadido anteriormente. Por último nos falta asignar desde la vista principal el *Delegate* del *Split View Controller* al *Detalle View Controller*, para hacer esto abrimos la vista `MainWindow.xib`, seleccionamos desde la lista de objetos el *Split View Controller* y relacionamos el Outlet "delegate" con el objeto *Detalle View Controller*, debe de quedar de la siguiente manera:



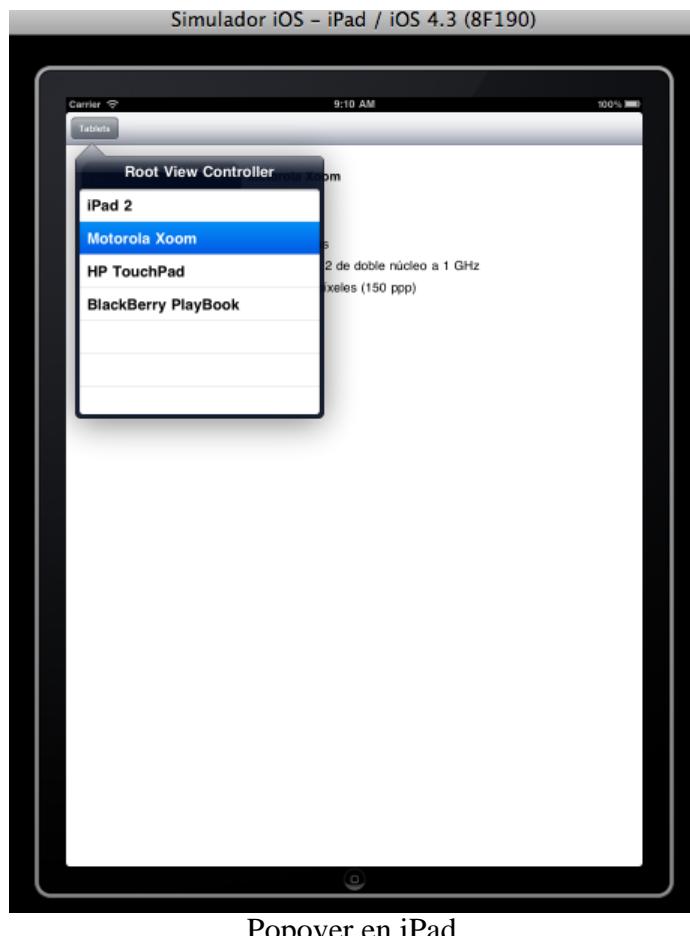
Vista general outlets

Si ejecutamos ahora la aplicación veremos que al girar el dispositivo y situarlo en

posición vertical aparecerá en la barra superior un botón que si lo pulsamos nos debe salir un *popover* con el listado de tablets. Como podemos ver, el tamaño del *popover* es muy alto y sobra casi todo, esto lo podemos evitar introduciendo el siguiente fragmento de código en el método `viewDidLoad` de la clase `MenuViewController`:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.clearsSelectionOnViewWillAppear = NO;
    self.contentSizeForViewInPopover = CGSizeMake(320.0, 300.0);
}
```

Volvemos a ejecutar la aplicación y comprobamos ahora que el tamaño del *popover* se ha reducido considerablemente.



Popover en iPad

13.1.2. Popovers

13.1.2.1. Introducción

Como hemos podido ver en el apartado anterior, un *popover* es una ventana secundaria (estilo *pupup*) que suele aparecer encima de la ventana principal de la interfaz. Se puede programar para que desaparezca automáticamente cuando el usuario toca cualquier parte de la pantalla o para que no desaparezca. El tamaño de la ventana *popover* puede cambiar también y por supuesto puede contener todo lo que se deseé, por ejemplo: un listado de opciones de configuración para nuestra aplicación, un álbum de fotos de la cual el usuario deberá seleccionar una, un formulario simple para que el usuario introduzca un texto, etc. En un iPhone, un *popover* puede corresponderse perfectamente con cualquier vista que ocupe la pantalla completa.

Con el fin de aclarar un poco más la definición de *popover* y ver cómo podemos utilizarlo en nuestras aplicaciones vamos a realizar un pequeño y sencillo ejemplo de uso partiendo del ejemplo del apartado anterior. Este ejemplo consistirá en mostrar un *popover* al pulsar un botón dentro de la vista de detalle. Este *popover* tendrá un pequeño listado de opciones que al seleccionarlas se mostrarán dentro de la vista de detalle. ¡Comenzamos!

13.1.2.2. Creando la clase delegada

Primero tenemos que crear la clase controladora que heredará de `UITableViewController`, esto lo hacemos pulsando en *New > New File*, seleccionamos `UIViewController Subclass, subclass of UITableViewController`. Importante marcar la opción de *Targeted for iPad* y desmarcar *With XIB for user interface*. Guardamos el fichero con el nombre `SeleccionaOpcionController`.

Ahora abrimos `SeleccionaOpcionController.h` y escribimos lo siguiente:

```
#import <UIKit/UIKit.h>

@protocol SeleccionaOpcionDelegate
- (void)opcionSeleccionada:(NSString *)nombreOpcion;
@end

@interface SeleccionaOpcionController : UITableViewController {
    NSMutableArray *_opciones;
    id<SeleccionaOpcionDelegate> _delegate;
}
@property (nonatomic, retain) NSMutableArray *opciones;
@property (nonatomic, assign) id<SeleccionaOpcionDelegate> delegate;
@end
```

En el fragmento de código anterior creamos un sólo método delegado que se encargará de notificar a otra clase cuando un usuario seleccione una opción del listado. A continuación abrimos `SeleccionaOpcionController.m` y hacemos los siguientes cambios:

```
// Debajo de @implementation
@synthesize opciones = _opciones;
@synthesize delegate = _delegate;

// Método viewDidLoad:
- (void)viewDidLoad {
    [super viewDidLoad];
    self.clearsSelectionOnViewWillAppear = NO;
    self.contentSizeForViewInPopover = CGSizeMake(150.0, 140.0);
    self.opciones = [NSMutableArray array];
    [_opciones addObject:@"Opción 1"];
    [_opciones addObject:@"Opción 2"];
    [_opciones addObject:@"Opción 3"];
}

// en dealloc:
self.opciones = nil;
self.delegate = nil;

// en numberOfSectionsInTableView:
return 1;

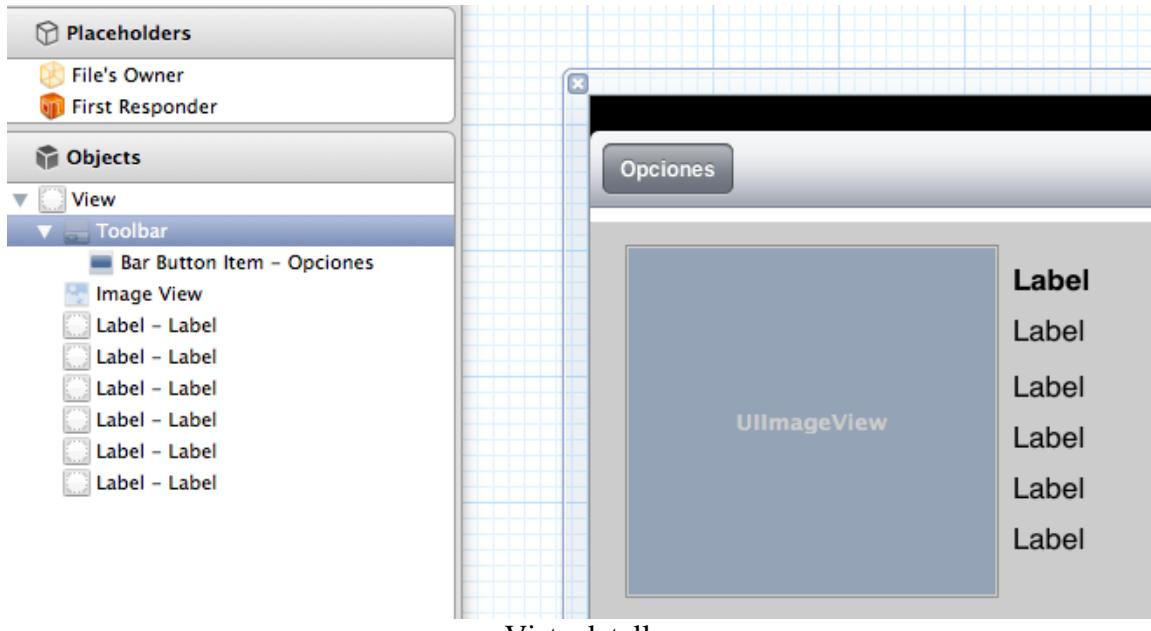
// en numberOfRowsInSection:
return [_opciones count];

// en cellForRowAtIndexPath:
NSString *opcion = [_opciones objectAtIndex:indexPath.row];
cell.textLabel.text = opcion;

// en didSelectRowAtIndexPath:
if (_delegate != nil) {
    NSString *opcion = [_opciones objectAtIndex:indexPath.row];
    [_delegate opcionSeleccionada:opcion];
}
```

13.1.2.3. Mostrando el popover

Una vez que hemos programado la clase delegada encargada de gestionar la selección de la opción sólo nos queda mostrar el *popover* dentro de la vista detalle de nuestra aplicación. Para ello abrimos la vista *DetalleViewController.xib*, añadimos un botón (*Bar Button Item*) al *toolbar* y le ponemos como título del botón: "Opciones". También añadimos un nuevo *Label* que será el que muestre la opción que seleccionemos de la lista.



Vista detalle

Ahora tenemos que programar el método que se llamará cuando se pulse sobre el botón y un par de variables necesarias:

DetalleViewController.h

```

        // debajo de los #import
#import "SeleccionaOpcionController.h"

// Añadimos el Delegate de selección de opción a la definición
@interface RightViewController : UIViewController
<TabletSelectionDelegate,
 UISplitViewControllerDelegate, SeleccionaOpcionDelegate> {

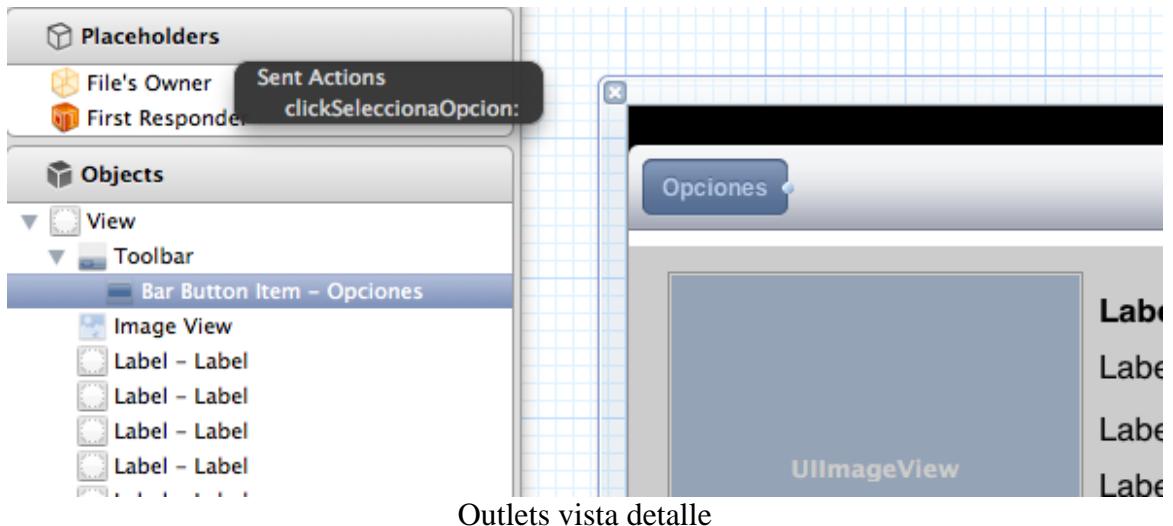
// Dentro de la clase añadimos estas dos variables
SeleccionaOpcionController *_seleccionaOpcion;
UIPopoverController *_seleccionaOpcionPopover;
UILabel *_labelOpcionSeleccionada;

// Añadimos estas tres propiedades
@property (nonatomic, retain) SeleccionaOpcionController
*seleccionaOpcion;
@property (nonatomic, retain) UIPopoverController
*seleccionaOpcionPopover;
@property (nonatomic, retain) IBOutlet UILabel *labelOpcionSeleccionada;

// Añadimos la definición del evento click del botón
-(IBAction) clickSeleccionaOpcion:(id) sender;

```

Ahora asignamos el Outlet del evento del botón dentro de la vista, *Ctrl+click* y arrastrar hasta File's Owner, ahí seleccionar "clickSeleccionaOpcion":



Outlets vista detalle

Nos queda completar el método de la clase delegada y el del evento click del botón. Editamos ahora `DetalleViewController.m`

```

// Añadimos los @synthesize nuevos
@synthesize seleccionaOpcion = _seleccionaOpcion;
@synthesize seleccionaOpcionPopover = _seleccionaOpcionPopover;
@synthesize labelOpcionSeleccionada = _labelOpcionSeleccionada;

// en dealloc
self.seleccionaOpcion = nil;
self.seleccionaOpcionPopover = nil;

// Añadimos estos métodos
- (void)opcionSeleccionada:(NSString *)nombreOpcion {

    //escribimos la opción seleccionada en la label
    self.labelOpcionSeleccionada.text = [NSString stringWithFormat:
        @"Has seleccionado: %@", nombreOpcion];

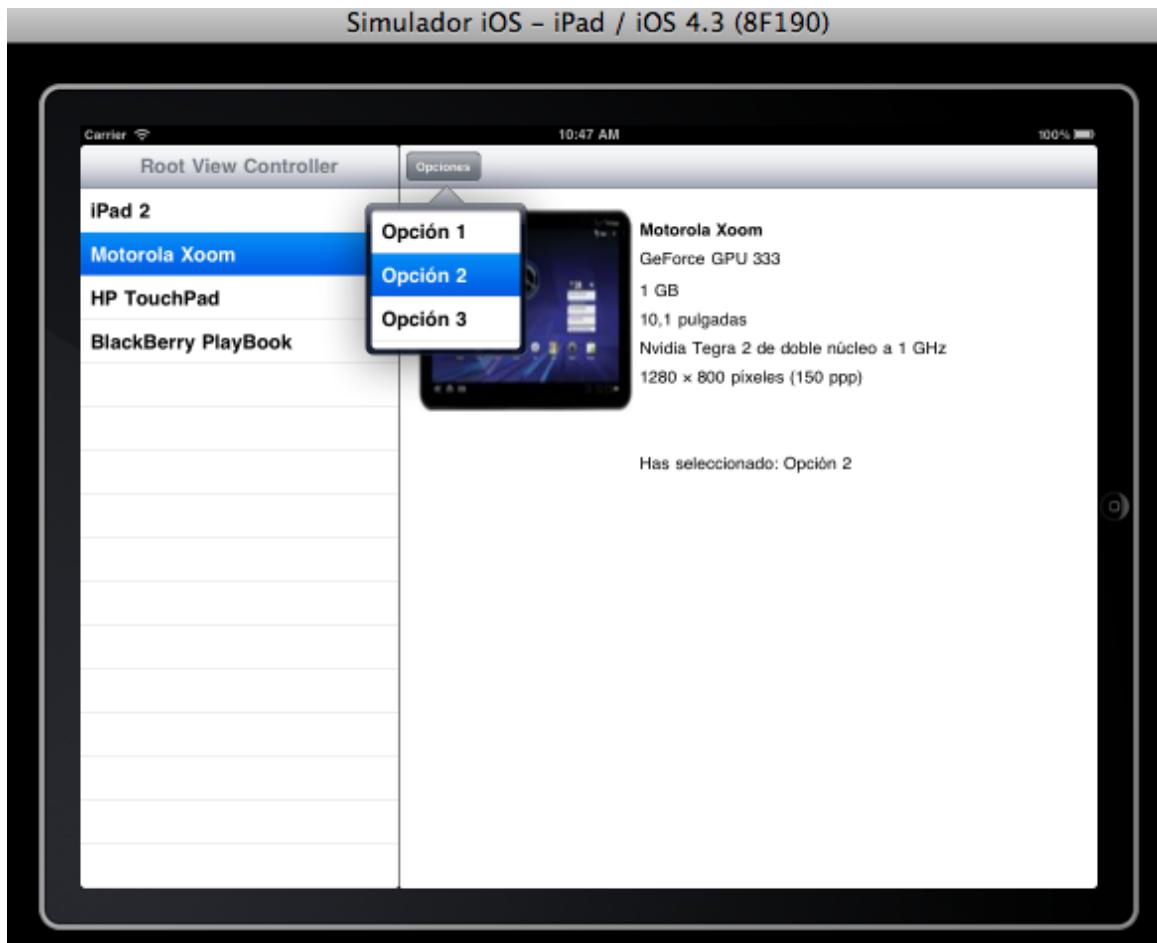
    //ocultamos el popover
    [self.seleccionaOpcionPopover dismissPopoverAnimated:YES];
}

- (IBAction)clickSeleccionaOpcion:(id)sender {
    if (_seleccionaOpcion == nil) {
        self.seleccionaOpcion = [[[SeleccionaOpcionController alloc]
            initWithStyle:UITableViewStylePlain] autorelease];
        _seleccionaOpcion.delegate = self;
        self.seleccionaOpcionPopover = [[[UIPopoverController alloc]
            initWithContentViewController:_seleccionaOpcion] autorelease];
    }
    [self.seleccionaOpcionPopover presentPopoverFromBarButtonItem:sender
        permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}

```

Ahora volvemos a abrir `DetalleViewController.xib` y relacionamos el label que hemos creado anteriormente con el outlet `labelOpcionSeleccionada`. Ejecutamos el

proyecto y comprobamos que todo funciona correctamente:



Popover en iPad

13.2. Aplicaciones universales

13.2.1. Introducción

En este apartado trataremos las **aplicaciones universales**, veremos qué ventajas e inconvenientes podemos encontrar a la hora de diseñarlas, cuál es el proceso de programación y las recomendaciones por parte de *Apple*.

Utilizamos el término **universal** para denominar a todas aquellas aplicaciones compatibles con todos los dispositivos iOS. Una aplicación universal podremos utilizarla tanto en iPhone como en iPad, sin necesidad de instalar dos distintas. Los usuarios agradecen enormemente este tipo de aplicaciones ya que mediante una sola compra pueden hacer uso en todos sus dispositivos. Por el lado de los desarrolladores no se tiene

tan claro si este tipo de aplicaciones son más rentables que realizar dos distintas a la hora de posicionarse mejor en la *App Store* y conseguir más ventas. Lo que sí está claro es que el desarrollo de una aplicación universal ahorra un tiempo de programación muy a tener en cuenta ya que programamos una sola aplicación y no dos. Otro punto a tener en cuenta a favor es a la hora de actualizar, ya que tendremos que preocuparnos de implementar los cambios sólo en una aplicación, no en dos. A la hora de implementar una aplicación universal tendremos que tener en cuenta si el usuario la está ejecutando sobre un iPhone / iPod Touch o sobre un iPad.



[Ver en iTunes](#)

⊕ Esta App se ha desarrollado tanto para iPhone como para iPad

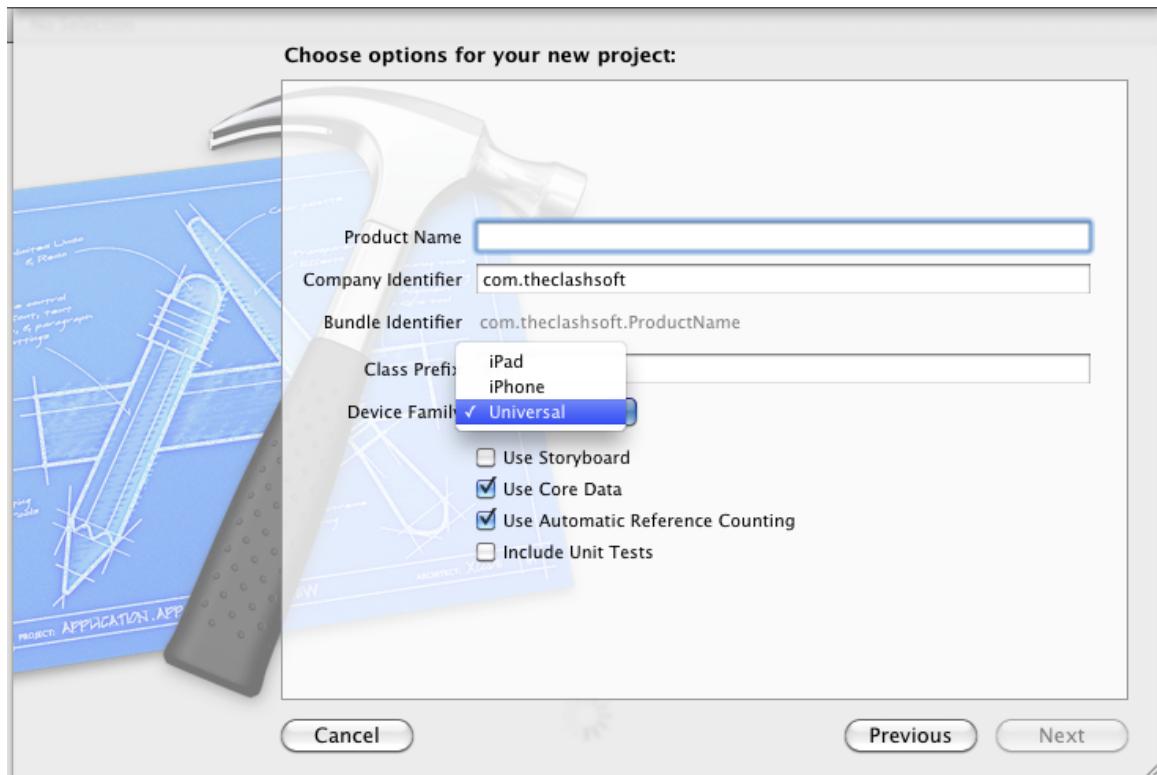
App universal UA

Con la llegada del iPad, el SDK de iPhone 3.2 empieza a soportar tres tipos de aplicaciones:

- **Aplicaciones iPhone.** Optimizadas para funcionar sobre un iPhone o iPod Touch. Estas aplicaciones se pueden ejecutar sobre un iPad en su resolución original o duplicando su resolución por 2 ajustándose de esta manera a la pantalla más grande.
- **Aplicaciones iPad.** Desde la versión de iOS 3.2, se pueden desarrollar aplicaciones optimizadas para iPad. Estas **sólo** funcionarán en iPad, no en otros dispositivos.
- **Aplicaciones Universales.** También desde la llegada de iOS 3.2 los desarrolladores pueden crear aplicaciones universales. Estas están optimizadas para funcionar en todos los dispositivos iOS. Es, en su esencia, una aplicación iPhone y una aplicación

iPad compiladas en un mismo binario.

A partir de la versión 4.2 de XCode podemos crear desde cero aplicaciones universales usando cualquier plantilla de las disponibles, utilizando esto ahorraremos mucho tiempo en cuanto a configuraciones y planteamientos iniciales. Entre las plantillas que encontramos en XCode destaca la ya mencionada anteriormente Master Detail Application, si la seleccionamos y marcamos la opción de *Universal* en Device Family, XCode nos creará la base para una aplicación universal empleando un controlador de tipo Split View, el cual hemos comentado en apartados anteriores.



Proyecto nuevo Universal

13.2.2. Diseñando la interfaz de una aplicación universal

El primer paso que debemos de realizar a la hora de hacer una aplicación universal desde cero es diseñar la interfaz para cada una de las vistas separando por un lado las vistas de iPhone/iPod Touch de las de iPad. A continuación detallaremos algunos de los puntos en los que deberemos pararnos a pensar a la hora de diseñar la interfaz:

- **Orientación.** Gracias al acelerómetro que viene en el hardware del dispositivo es posible detectar la orientación actual. Con esta información podremos adaptar la interfaz de la aplicación para acomodarla a la orientación que tenga el dispositivo en un momento dado. En aplicaciones iPhone, la adaptación de la interfaz no es tan

importante como en aplicaciones para iPad. Haciendo uso en la medida de lo posible de esta funcionalidad mejoraremos considerablemente la experiencia de usuario.

- **Estructura de vistas.** La pantalla del iPad, al ser más grande y de mayor resolución que la del iPhone permite al usuario acceder a más información en un mismo lugar.
- **Gestos.** En la pantalla del iPad podremos realizar muchos más gestos que en la del iPhone debido al tamaño de su pantalla, por ejemplo gestos con cuatro dedos al mismo tiempo.
- **Vistas partidas (Split Views).** Como hemos explicado anteriormente, una vista partida o *Split View* es una estructura de ventanas única en iPad. Mediante esta podremos dividir la pantalla del iPad en dos, mostrando distinta información en cada una de las vistas y mejorar de esta manera la usabilidad y la experiencia de uso.
- **Ventas emergentes (Popovers).** Es otra de las funcionalidades disponibles únicamente en iPad. Mediante este tipo de ventanas podremos mostrar listados o distintas opciones en un momento determinado dentro de la ejecución de nuestra aplicación.
- **Características Hardware.** Distintos dispositivos iOS poseen distintas funcionalidades hardware, unos tienen cámara de fotos, otros tienen brújula, etc. Como es lógico, hay que tener en cuenta la ausencia de este tipo de características a la hora de implementar una aplicación universal.

En la documentación que nos facilita *Apple* podremos encontrar toda la información detallada sobre las aplicaciones universales así como una extensa guía de compatibilidades entre dispositivos iOS. Podrás acceder desde [esta url](#).

13.2.3. Programando una aplicación universal

A la hora de programar una aplicación universal en iOS tendremos que tener en cuenta la disponibilidad de las características de nuestra aplicación cuando se esté ejecutando en todos los dispositivos, esto lo programamos mediante *código condicional*. A continuación detallamos una serie de categorías de funcionalidades en las que tendremos que pensar a la hora de programar una aplicación universal:

- **Recursos.** En una aplicación universal necesitarás distinguir los ficheros de interfaz (nibs) a utilizar según la plataforma en donde se esté ejecutando la aplicación. Normalmente necesitarás dos ficheros distintos para cada vista o controladora, una para iPad y otra para iPhone. En cuanto a las imágenes o gráficos a utilizar dentro de la aplicación normalmente tendrás que distinguirlos también según el tipo de dispositivo, por ejemplo, si se está ejecutando sobre un iPad las imágenes deberán tener mayor resolución que si se está ejecutando sobre iPhone/iPod Touch. En el caso de iPhone 4/4S, con la pantalla *retina display*, se pueden utilizar las mismas imágenes en alta resolución que en el iPad.
- **Nuevas APIs.** Existen APIs destinadas únicamente para iPhone y otras para iPad. Por ejemplo, la librería `UIGraphics` para iPad soporta PDFs. El código debe de comprobar si estamos ejecutando la aplicación es un iPhone o en un iPad ya que los métodos correspondientes a la gestión de los PDFs sólo estarán disponibles para este

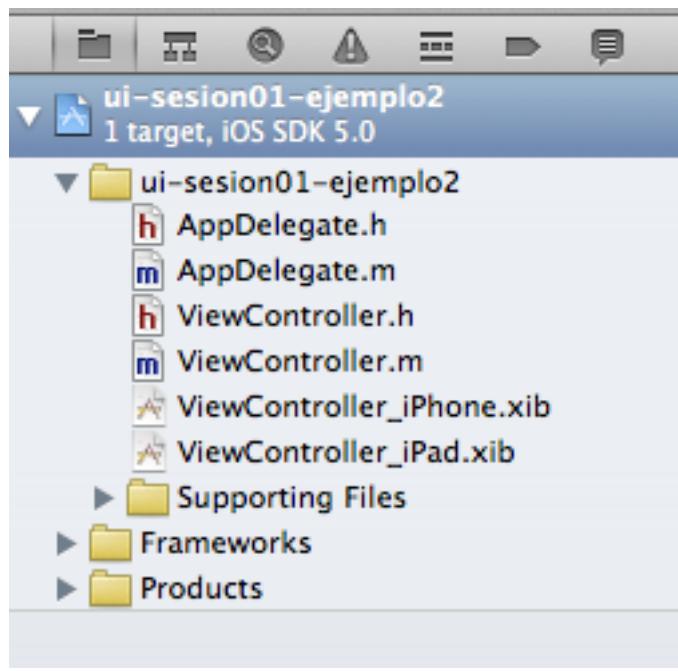
último.

- **Capacidades Hardware.** A veces es necesario detectar si ciertas capacidades hardware están disponibles en el dispositivo en el que estamos ejecutando la aplicación, por ejemplo a la hora de usar la cámara ya que hay dispositivos que no la tienen.

Para entender mejor en qué consiste una aplicación universal y de cómo podemos comenzar a programar una, vamos a realizar un sencillo ejemplo paso a paso en el que mostraremos los detalles de una película en una vista. Por el tamaño de la información a mostrar sobre el libro tendremos que crear **dos vistas distintas**: por un lado una adaptada al tamaño de pantalla de un iPhone/iPod Touch y por otro lado otra adaptada a un iPad.

Comenzamos abriendo *XCode* y creando un nuevo proyecto de tipo *Single View Application* al que llamaremos *ui-sesion01-ejemplo2*. Del resto de opciones seleccionamos el tipo de aplicación *Universal* (esto también se puede cambiar más adelante en el apartado "Summary" del proyecto) y desmarcamos *Use Storyboard*, pusamos sobre "Next" y guardamos el proyecto.

Como podemos ver, *XCode* automáticamente nos ha creado la estructura inicial básica para poder comenzar con el desarrollo de una aplicación universal. Si nos fijamos en el árbol de ficheros veremos que, aparte de la clase delegada (*AppDelegate*) tenemos un *View Controller* y dos vistas del *Interface Builder* que hacen uso este mismo *View Controller*: una vista con diseño para iPhone (*ViewController_iPhone.xib*) y otra para iPad (*ViewController_iPad.xib*).



Arbol proyecto universal

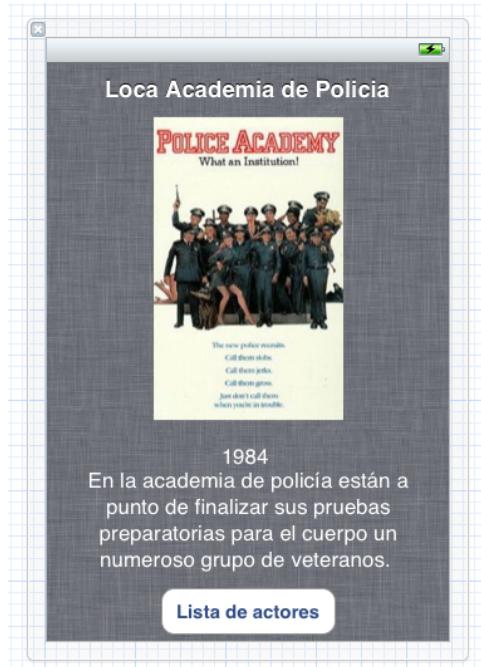
Vamos a analizar ahora en detalle la clase `AppDelegate`, en ella podemos ver como dentro del método `didFinishLaunchingWithOptions` cargamos una vista u otra dependiendo del tipo de dispositivo que esté ejecutando la aplicación. Esto se hace mediante el `userInterfaceIdiom` de la clase `UIDevice`:

```
if ([[UIDevice currentDevice] userInterfaceIdiom] ==  
UIUserInterfaceIdiomPhone) {  
    self.viewController = [[[ViewController alloc]  
    initWithNibName:@"ViewController_iPhone" bundle:nil] autorelease];  
} else {  
    self.viewController = [[[ViewController alloc]  
    initWithNibName:@"ViewController_iPad" bundle:nil] autorelease];  
}
```

Como podemos ver en el fragmento de código anterior, si el dispositivo es *iPhone* cargamos la controladora *View Controller* con la vista `ViewController_iPhone`; en caso contrario cargamos el mismo controlador pero con la vista para *iPad* `ViewController_iPad`. De este modo estamos ejecutando la misma controladora pero con distintas vistas dependiendo del dispositivo que se utilice. Entre muchas de las ventajas que encontramos usando este método es que sólo deberemos programar y mantener una sola clase (en este caso `ViewController`), la cual estará "asociada" a dos vistas distintas.

Para completar el ejemplo vamos a "rediseñar" las vistas de modo que queden de la siguiente manera:

`ViewController_iPhone.xib`



Vista iPhone

ViewController_iPad.xib



Vista iPad

Una vez diseñadas las vistas vamos a ejecutar la aplicación usando primero el simulador de iPhone y luego el de iPad. Veremos que funciona como esperamos, las vistas se cargan según el tipo de dispositivo que hayamos indicado:

Vista en iPad



Detalle iPad

Una vez estudiado la gestión básica de vistas en aplicaciones universales vamos a comprobar que la gestión de tablas (`UITableView`) es aún más simple. Para ello vamos a terminar este sencillo ejemplo creando la vista del listado de actores de nuestra película. Para hacer esto vamos a añadir una nueva clase a nuestro proyecto de tipo `UIViewController`. Pulsamos sobre *File > New > New File* seleccionando *iOS/Cocoa Touch, UIViewController Subclass*. En el campo de *Class* escribimos "**ListaActoresTableView**" y en *Subclass of* seleccionamos `UITableViewController`. Dejamos desmarcada la opción de "Targeted for iPad" y marcamos "With XIB for user interface".

Abrimos el fichero `ListaActoresTableView.h` y definimos un objeto de tipo `NSMutableArray` que contendrá el listado de actores que queremos mostrar en la tabla:

```
#import <UIKit/UIKit.h>

@interface ListaActoresTableView : UITableViewController

@property (strong, nonatomic) NSMutableArray *listaActores;

@end
```

Ahora en archivo `ListaActoresTableView.m` escribimos la variable con el prefijo `@synthesize` e inicializamos el array. Completamos también las funciones del protocolo de `UITableView Controller`. El código quedaría de la siguiente manera:

```
@synthesize listaActores = _listaActores;
```

```

    - (void)viewDidLoad
{
    [super viewDidLoad];

    self.listaActores = [[NSMutableArray alloc] init];
    [self.listaActores addObject:@"Steve Guttenberg"];
    [self.listaActores addObject:@"Kim Cattrall"];
    [self.listaActores addObject:@"G.W. Bailey"];
    [self.listaActores addObject:@"Bubba Smith"];
    [self.listaActores addObject:@"Donovan Scott"];
    [self.listaActores addObject:@"George Gaynes"];
}

    - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return [self.listaActores count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CellIdentifier] autorelease];
    }

    // Configure the cell...
    cell.textLabel.text = [self.listaActores objectAtIndex:indexPath.row];
    return cell;
}

```

Por último nos falta enlazar el botón de la primera vista para que enlace con este controlador, para ello creamos una acción para el botón dentro de la clase `ViewController`. Añadimos la definición del método en el fichero `.h` y después desde la vista lo enlazamos para que "salte" con el evento "Touch Up Inside". Esto último deberemos hacerlo en las dos vistas (la del iPhone y la del iPad). Este es el método que implementa la acción del botón y que abre en un *Modal View* la vista de la lista de actores que hemos creado anteriormente:

```

-(IBAction)abreVistaActores:(id)sender {
    ListaActoresTableView *listaActoresTableView = [[ListaActoresTableView

```

```
alloc]
    initWithNibName:@"listaActoresTableView" bundle:nil];
    [self presentModalViewController:listaActoresTableView animated:YES];
}
```

Una vez programado esto último podemos ejecutar de nuevo la aplicación y probarla en iPhone y en iPad. Podemos ver que en el caso de las tablas, estas se adaptan al tamaño de la pantalla evitándonos de esta forma crear dos vistas distintas como hemos comentado al principio del ejercicio. En el caso que queramos profundizar más en el diseño posiblemente tendremos que crear dos vistas distintas, pero como base se puede emplear una única.

Vista en iPad

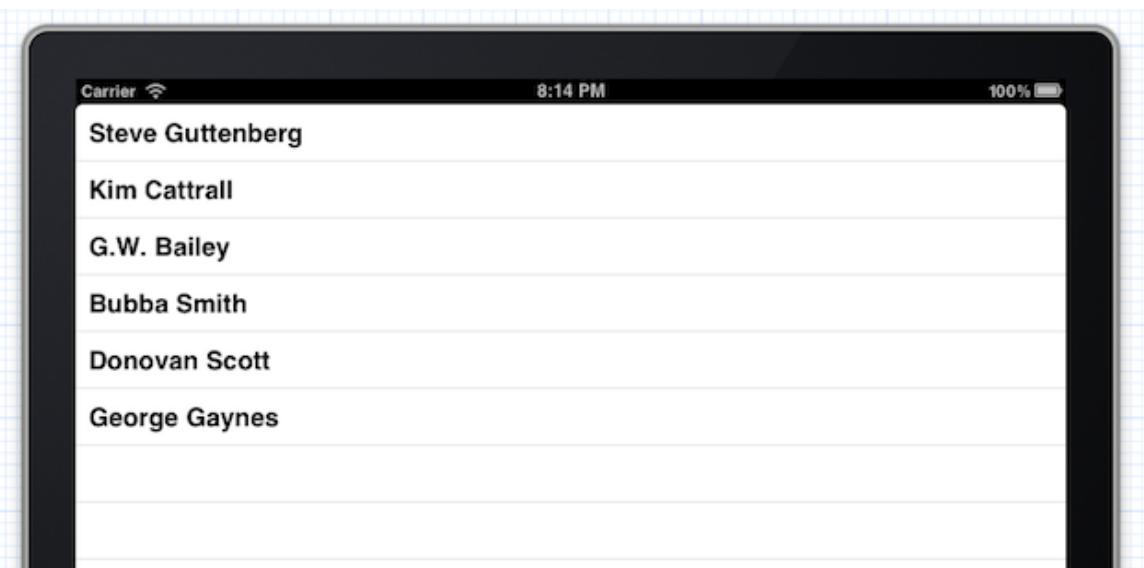


Table View iPad

14. Componentes para iPad y aplicaciones universales - Ejercicios

14.1. Programando una aplicación para iPad con Split View (1)

- a) Vamos a crear un nuevo proyecto de tipo *Single View Application*. El nombre del proyecto será `BibliotecaMusical` y nuestra organización `es.ua.jtech`. Utilizaremos como prefijo para todas las clases `UA` y será sólo para **iPad**. También desmarcaremos todas las casillas adicionales, ya que no vamos a utilizar por el momento ninguna de esas características (Core Data, ARC, pruebas de unidad, storyboards).
- b) Ahora añadimos un controlador de tipo `Split View Controller` a la vista `UAVViewController.xib`. ¿Qué dos vistas forman la vista *Split View*?
- c) Creamos la vista de la izquierda (`UITableViewController`). Creamos una nueva clase que llamaremos `ListadoDiscosViewController` la cual será de tipo `UIViewController subclass` y heredará de (`subclass of`) `UITableViewController`. Recuerda marcar las casillas de *Targeted for iPad* y *With XIB for user interface*
- d) Creamos la vista de la derecha, para ello añadiremos un nuevo archivo a nuestro proyecto que se llamará `DetalleDiscoViewController` la cual será de tipo `UIViewController subclass` y heredará de `UIViewController`. Como en el paso anterior dejaremos marcadas las opciones de *Targeted for iPad* y de *With XIB for user interface*.
- e) Añadimos el código necesario dentro de la clase `didFinishLaunchingWithOptions` del fichero `UAAAppDelegate` que declare un `SplitViewController` y asigne las clases del listado de discos y del detalle a este.
- f) Arrancamos la aplicación y comprobamos que funciona bien. ¿Qué vista corresponde a la "Master" y cual al "Detail"? ¿Qué pasaría si no asignamos la vista detalle al Split View? ¿Y si no asignamos la vista Master?
- g) Modificamos la clase `ListadoDiscosViewController` para que muestre 4 títulos de discos. Deberemos modificar los métodos necesarios de la clase `UITableViewController`.
- h) Modificamos la clase `DetalleDiscoViewController` para que muestre una label para el título del álbum seleccionado, otra label para el grupo/autor, otra para el número de temas y por último otra para la carátula/títulos de discos. Deberemos modificar la vista de la clase `DetalleDiscoViewController`, añadir los Outlets necesarios y enlazarlos correctamente.

14.2. Programando una aplicación para iPad con Split View (2)

Continuamos con el ejercicio anterior, ahora debemos enlazar la vista de la izquierda con la de la derecha. Cuando seleccionemos un álbum, la vista de detalle (derecha) debe de mostrar todos los datos de dicho álbum en sus correspondientes labels (creadas en el ejercicio anterior).

14.3. Programando una aplicación para iPad con Split View usando XCode 4.2

En este ejercicio vamos a diseñar la misma aplicación que hemos programado anteriormente pero usando las últimas opciones presentadas en la última versión de XCode. Para ello simplemente deberemos crear un nuevo proyecto al que llamaremos BibliotecaMusical 2 usando la plantilla Master-Detail View Controller. Arranca la aplicación. ¿Qué diferencias encuentras respecto al proyecto de BibliotecaMusical que hemos creado anteriormente?

14.4. Programando una aplicación universal

- a) Vamos a crear ahora una aplicación universal desde cero. Creamos un nuevo proyecto en XCode que llamaremos UnaPeliculaCualquiera. El proyecto va a ser de tipo *Single View Application*, organización es.ua.jtech y prefijo UA. Seleccionamos iPhone como "target" y desmarcaremos todas las casillas adicionales, ya que no vamos a utilizar por el momento ninguna de esas características (Core Data, ARC, pruebas de unidad, storyboards).
- b) Modificamos la vista UAVViewController.xib añadiendo una label para el título de la película, otro para la sinopsis y una imagen para la carátula (UIImageView). Creamos las propiedades y Outlets correspondientes en la clase UAVViewController.
- c) Creamos ahora la vista para iPad que llamaremos UAVViewController_iPad y añadimos las mismas etiquetas que en la vista de iPhone pero organizadas de distinta manera. Modificamos la propiedad del *Identity Inspector* necesaria para que la vista cargue los Outlets de la controladora y asignamos los Outlets necesarios para que la clase sea *key-value compliant*.
- d) Añadimos el código necesario dentro del método `didFinishLaunchingWithOptions` que cargue una vista u otra dependiendo del tipo de dispositivo que esté ejecutando la aplicación.
- e) ¿Qué propiedad del proyecto deberemos de modificar para que la aplicación se considere como "Universal"? Cambia dicha propiedad.
- f) Arrancamos la aplicación en ambos dispositivos en el simulador y comprobamos que funciona todo bien. ¿Y si queremos añadir una vista de tabla? ¿Tendremos que hacerlo en dos vistas por separado o no hace falta?

14.5. Programando una aplicación para iPad con un Popover

- a) En este ejercicio vamos a mostrar los detalles básicos de una película cualquiera dentro de un popover. Comenzamos creando un nuevo proyecto en XCode que llamaremos PeliculaEnPopover. El proyecto va a ser de tipo *Single View Application*, organización es.ua.jtech y prefijo UA. Seleccionamos iPad como "target" y desmarcaremos todas las casillas adicionales, ya que no vamos a utilizar por el momento ninguna de esas características (Core Data, ARC, pruebas de unidad, storyboards).
- b) Editamos la vista UAVViewController.xib y añadimos al menos dos carátulas de películas dentro de botones (UIButton).
- c) Creamos una clase nueva que será la que muestre el popover. Esta clase la llamaremos DatosPelicula y será de tipo UIViewController.
- d) Modificamos la vista de la clase DatosPelicula como queramos, añadiendo al menos dos labels. Añadimos las propiedades y Outlets necesarios a la clase.
- e) Implementamos las acciones de los botones de la clase UAVViewController y las asignamos en la vista.
- f) Dentro de las acciones de los botones crearemos un popover y lo mostraremos de modo que la flecha apunte al botón. Los objetos de tipo UIPopoverController deben de crearse como propiedades dentro de la clase UAVViewController y tendremos que realizar las comprobaciones necesarias para que no existan dos popover iguales al mismo tiempo en pantalla.

15. Guías de estilo y personalizaciones avanzadas

En esta sesión hablaremos de los patrones de diseño que *Apple* nos recomienda seguir para nuestras aplicaciones. En muchos casos estos patrones son obligatorios cumplirlos a la hora de diseñar nuestras aplicaciones y el hecho de no cumplirlos puede ser motivo de rechazo a la hora de querer publicar en la *App Store*. Comentaremos las distintas características de cada plataforma iOS. Por último detallaremos distintas técnicas que existen para personalizar los controladores y vistas más usados y, de esta forma, conseguir un aspecto más atractivo para el usuario final. ¡Comenzamos!

15.1. Guías de estilo en iOS

Tanto el iPhone como el iPad han supuesto un revolucionario giro en cuanto a diseño de interfaz se refiere. Los usuarios de este tipo de dispositivos pueden realizar múltiples tareas como navegar por internet, leer y escribir correos, realizar fotos y videos, navegar por las distintas aplicaciones, etc. Para evitar el "caos" a la hora de diseñar cualquier tipo de aplicación, en *Apple* han planteado una serie de guías de estilo que todo diseñador debe seguir en la medida de lo posible a la hora de diseñar una aplicación. Estas guías de estilo no son válidas, como es lógico, a la hora de diseñar la interfaz de un juego, sólo son útiles en aplicaciones.

En este módulo vamos a aprender a diseñar aplicaciones de iPhone / iPad que sean usables, accesibles y en las que el usuario perciba una grata experiencia al hacer uso de ella. Hay que tener en cuenta que un usuario de iPhone está habituado a una serie de elementos y situaciones básicas como la navegación dentro de vistas mediante *Navigation Controllers*, navegación por las opciones fundamentales mediante los *Tab Bar Controllers*, uso de botones claros, etc. Aquí haremos un resumen de todo lo que [Apple propone](#) en sus guías de estilo.

Vamos a dividir el módulo en 3 puntos claramente diferenciados:

- Características principales de la plataforma iOS
- Estrategias de diseño de aplicaciones
- Guías de uso de las principales tecnologías disponibles en iOS

15.1.1. La pantalla

Todo usuario que usa una aplicación accede mediante la pantalla, es por ello que este es el componente más importante que podemos encontrar en cualquier dispositivo iOS. Existen tres distintos tamaños de pantalla según cada dispositivo, estos son:

- *iPhone 4 (Retina Display)*: 640 x 960 pixels
- *iPad*: 768 x 1024 pixels
- *iPhone 3G, 3GS o iPod Touch*: 320 x 480 pixels

Detección de contacto (touch events)

El tamaño de la zona mínima para que el evento de contacto funcione correctamente debe de ser de al menos 44 x 44 puntos.

15.1.2. Orientación del dispositivo

Uno de los requisitos a la hora de publicar una aplicación de iPhone/iPad en *App Store* es la compatibilidad con las distintas posiciones que puede adoptar el dispositivo iOS, estas posiciones son vertical (*portrait*) y horizontal (*landscape*). Cualquier aplicación debe de estar adaptada a ambas posiciones y esto se hace para mejorar la usabilidad y la comodidad a la hora de utilizar la aplicación. La programación de la orientación en una aplicación iOS es relativamente sencilla, excepto casos puntuales y no implementarla puede suponer, como hemos comentado, el rechazo a la hora de publicar en *App Store*.

Según el dispositivo que dispongamos, la pantalla inicial (*Home Screen*) acepta una orientación vertical, horizontal o ambas. Por ejemplo, un iPhone o iPod Touch muestra su pantalla de inicio sólo en vertical, mientras que en el iPad se puede mostrar en vertical y en horizontal.

15.1.3. Los multigestos

Un punto que hay que tener muy presente a la hora de diseñar y programar una aplicación en cualquier dispositivo iOS es el uso de los gestos, llamamos "gestos" a los distintos movimientos que hace el usuario sobre la pantalla para realizar distintas acciones. Cuantos más gestos se implementen en una aplicación, más agradable e intuitiva será de usar para el usuario final. Los distintos gestos que una aplicación puede adoptar son los siguientes:

- Toque (*tap*): Consiste en presionar o hacer "click" sobre un botón o cualquier objeto que esté en pantalla.
- Arrastre (*drag*): Mover el pulgar sobre la pantalla en una dirección, puede usarse para navegar sobre los elementos de una tabla, por ejemplo.
- Arrastre rápido (*flick*): Como el anterior, pero más rápido. Sirve para moverse por la pantalla de forma rápida.
- Arrastre lateral (*Swipe*): Mover el pulgar en dirección horizontal, sirve para mostrar el botón de "Eliminar" en una fila de una tabla.
- Doble toque (*double tap*): Presionar dos veces seguidas y de forma rápida la pantalla. Sirve para aumentar una imagen o un mapa, por ejemplo.
- Pellizco exterior (*pinch open*): Gesto de pellizco sobre la pantalla que sirve para aumentar una imagen o un mapa.
- Pellizco interior (*pinch close*): El inverso del anterior.
- Toque continuo (*touch and hold*): Como el toque básico, pero manteniendo el pulgar sobre la pantalla. Sirve para mostrar un menu contextual sobre la zona que se pulsa.

15.1.4. La multitarea

La multitarea es una característica que está disponible a partir de iOS 4. Mediante esta nuestro dispositivo puede ejecutar más de dos aplicaciones al mismo tiempo: siempre estará una ejecutándose en pantalla, mientras que el resto estarán almacenadas en memoria, en *background*. Apple nos recomienda que nuestra aplicación tenga dicha característica ya que de esta forma el usuario puede estar realizando otras tareas al mismo tiempos sin necesidad de cerrarla.



Captura multitarea iOS

La gestión e implementación de la multitarea es muy sencilla, se realizará desde la clase delegada, en los métodos `applicationDidEnterBackground` y `applicationWillEnterForeground`. Esto se explicará en las sesiones correspondientes.

15.1.5. Preferencias y ayuda

Si la aplicación utiliza preferencias propias, estas deben de estar en la medida de lo posible en el panel de preferencias que se encuentra en el menu de general de configuración de la aplicación de preferencias del dispositivo.

Por otro lado, la el texto de ayuda, si existe, debe de ser lo más claro y compacto posible,

si se pueden utilizar imágenes mejor. Hay que tener en cuenta que el usuario no tendrá tiempo ni ganas de estar leyendo la ayuda antes de utilizar tu aplicación, además el espacio que ocupa la ayuda se puede emplear para otros contenidos que sean de mayor importancia. La ayuda no debería ser necesaria si se utilizan las guías de estilo establecidas por *Apple* y que tienen por finalidad, como hemos comentado anteriormente, establecer una interfaz simple e intuitiva de usar.

15.1.6. Estrategias de diseño: Documento descriptivo de la aplicación

Existen distintas estrategias de diseño de aplicaciones. *Apple* nos recomienda redactar un pequeño documento que describa la aplicación de la manera más clara y concisa posible siguiendo cuatro pasos simples:

- 1) Listar todas las tareas que debe implementar la aplicación sin importar que esta lista sea grande, después puede reducirse.
- 2) Determinar quiénes son los usuarios que van a usar la aplicación.
- 3) Filtrar el listado de tareas del primer punto según los tipos de usuario definidos en el segundo punto.
- 4) Usar el listado de tareas final para definir el tipo de interfaz gráfica a utilizar, los controles y terminología, etc.

15.1.7. Estrategias de diseño: Diseño según el dispositivo

La aplicación que diseñas tiene que estar totalmente adaptada a un dispositivo iOS, no a otro tipo ni a Web, el usuario lo agradecerá. La inmensa mayoría de usuarios de iOS están acostumbrados al uso de botones, barras de navegación, *Tab Bars*, etc. Debemos de, en la medida de lo posible, hacer uso de toda esta serie de componentes que son diseñados de forma específica para dispositivos iOS y con los que los usuarios están muy familiarizados.

Otro tema importante es que si la aplicación es universal, esta debe de funcionar correctamente tanto en iPhone como en iPad. Hay que asegurarse antes de publicarla en *App Store* de esto ya que si se llega a publicar y falla en alguno de estos dispositivos, el usuario lo tendrá en cuenta a la hora de puntuarla o comprar otras del mismo desarrollador. Existen una serie de puntos a tener en cuenta en este caso:

- Hay que "moldear" cada aplicación según el dispositivo en el que se use teniendo en cuenta especialmente las capas de vistas, ya que estas cambian bastante.
- Se debe de adaptar todo el arte al dispositivo adecuado. El iPad y el iPhone 4 Retina display tendrá un arte con mayor resolución que un iPhone 4, 3G, 3GS y iPod Touch.
- Las características de la aplicación deben de conservarse a pesar del tipo de dispositivo que se use.
- Intentar, al menos, diseñar la aplicación universal y no únicamente para iPhone y que por defecto aparezca aumentada 2x en iPad. El usuario lo agracederá.

15.1.8. Guías de uso de las principales tecnologías disponibles en iOS

La API de iOS nos da acceso a múltiples tecnologías nativas que nos permiten realizar distintas tareas dignas de destacar y que posiblemente sean de utilidad a la hora de usar la aplicación que desarrollemos. A continuación se comentan las principales:

15.1.8.1. Multitarea

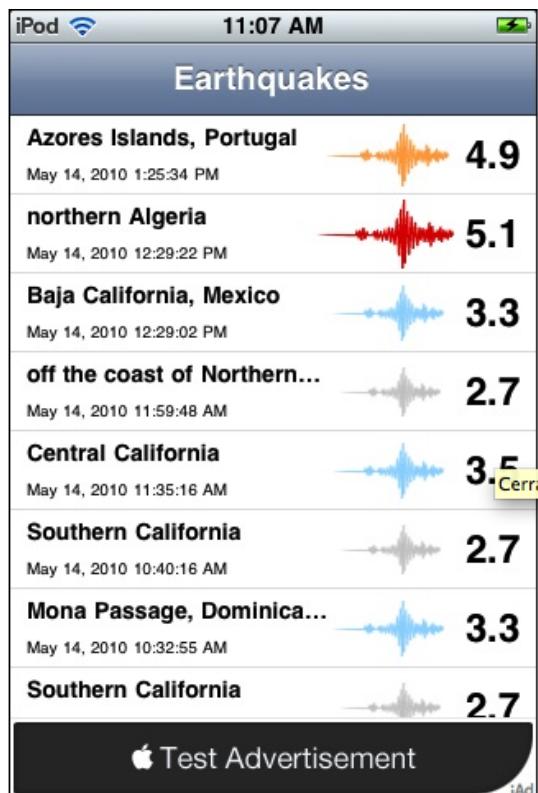
La multitarea aparece a partir de la versión de iOS 4, es muy importante tener en cuenta esta característica a la hora de desarrollar una aplicación ya que el usuario en cualquier momento puede cambiar entre ellas y si no está contemplado puede llegar a producir errores o inconsistencias. La aplicación debe de estar preparada para gestionar interrupciones de audio en cualquier momento, pararse y reiniciarse sin ninguna complicación ni lag y de forma "suave", por último debe comportarse de forma adecuada cuando se encuentra en *background*. La multitarea es una tecnología que se utiliza muy a menudo a la hora de usar aplicaciones iOS y es por ello que debemos tenerla muy presente a la hora de programar nuestras aplicaciones. La gestión del paso de un estado *activo* a *inactivo* se suele programar dentro de la clase *Delegada*.

15.1.8.2. Imprimir

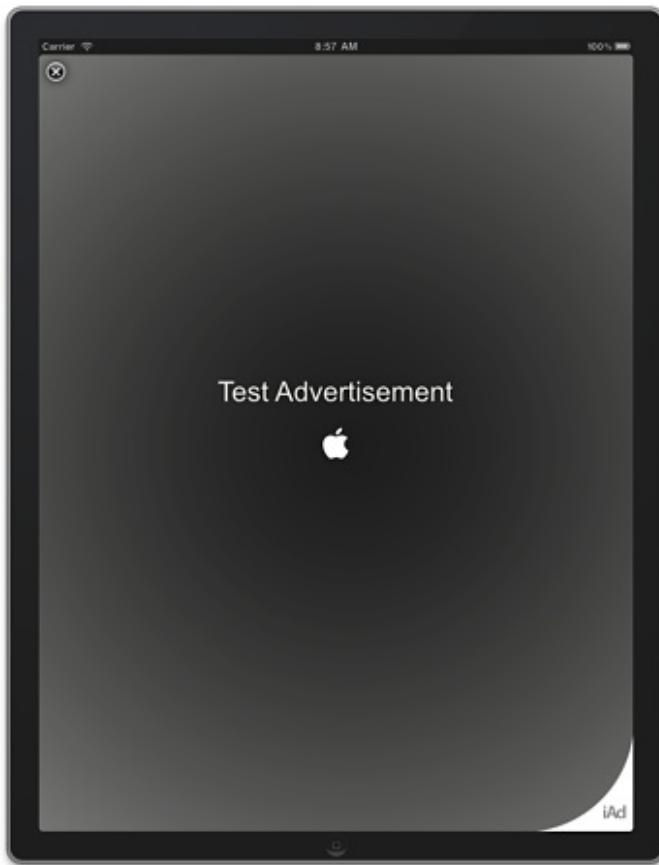
A partir de la versión iOS 4.2 aparece la tecnología de imprimir. Mediante esta ya se puede enviar a imprimir cualquier documento que deseemos desde nuestras aplicaciones. La API de iOS se encarga de gestionar la localización de impresoras y la ejecución de tareas de impresión, el desarrollador se encarga de especificar qué elemento/s desea imprimir (normalmente PDF o imágenes) y gestionar el botón de *imprimir*.

15.1.8.3. iAD

A partir de iOS 4.0 se pueden mostrar banners de publicidad gestionados por *Apple* en nuestras aplicaciones. Existen varias formas y tamaño de mostrar estos banners dependiendo del dispositivo que utilicemos. El sistema iAD es un método a tener en cuenta para obtener beneficios extra y se utiliza bastante a menudo. La publicidad de iAD no es intrusiva, cuando un usuario pulsa sobre uno de estos banners, la aplicación se debe de *pausar* y la publicidad se abrirá en una ventana dentro de la aplicación. Cuando el usuario cierra esta ventana, la aplicación que se estaba usando seguirá desde el mismo punto donde se quedó. Esta característica le distingue enormemente de otros sistemas de publicidad disponibles en iOS como el famoso *AdMob* de *Google*. A continuación muestro una serie de ejemplos de este tipo de publicidad:



Ejemplo iAD



Ejemplo iAD pantalla completa iPad

15.1.8.4. Vista rápida de documentos

Desde la versión 4 de iOS los usuarios pueden acceder a una vista previa de documentos descargados desde las aplicaciones. Dependiendo del dispositivo a utilizar, los documentos se presentarán dentro de una ventana modal si estamos hablando de un iPad o dentro de una vista completa si es un iPhone/iPod Touch.

15.1.8.5. Sonidos

Puede que como desarrollador estés interesado en reproducir sonidos dentro de tu aplicación, esto puede producir un efecto agradable en el usuario aunque hay que tener en cuenta una serie de puntos para evitar justamente lo contrario. Si el usuario activa el modo *silencio* en el dispositivo, este no debe de emitir ningún tipo de sonido, por lo tanto esto es algo a tener muy en cuenta cuando se desarrolle una aplicación con sonido. En principio no debe haber problemas, pero conviene comprobarlo siempre. En la medida de lo posible, dejar al usuario la opción de ajustar los volúmenes de sonido que desee. El tipo de sonido a escoger a la hora de desarrollar las distintas características de la

aplicación es muy importante, estos sonidos deben de ser los más adecuados y se deben de ajustar al tipo de acción que se esté desarrollando. Otro punto a tener en cuenta es la gestión de las interrupciones de audio cuando, por ejemplo, se produce una llamada entrante o la aplicación pasa a estado de inactiva (*background*), etc. Si estamos hablando de una aplicación musical podemos tener en cuenta, para mejorar la experiencia de usuario, el uso de control remoto de *Apple* o incluso el nuevo sistema de *AirPlay*.

15.1.8.6. Accesibilidad

El sistema de control por voz *VoiceOver* está diseñado para aumentar la accesibilidad a personas con discapacidades visuales o incluso para usuarios con interés en el aprendizaje de una lengua extranjera.

15.1.8.7. Menú de edición

Existe un menú contextual de edición básico que incorpora iOS por defecto, este contiene las opciones de "copiar", "cortar" y "seleccionar". Este menú los desarrolladores podemos modificarlo a nuestro antojo eliminando o incluyendo nuevas opciones que se ajusten más a nuestra aplicación.

15.1.8.8. Teclado

El teclado virtual es totalmente modificable según nuestros requerimientos, por ejemplo, si queremos utilizar un teclado numérico sólo debermos indicarlo al cargar el teclado dentro del método adecuado.

15.1.8.9. Servicios de localización

Los servicios de localización permiten a los usuarios acceder a una situación geográfica en cualquier momento y lugar. Si nuestra aplicación hace uso de los mapas, por ejemplo, estos intentarán a su vez hacer uso de los servicios de localización del dispositivo para poder situar de una forma bastante certera la posición actual del usuario en el mapa. Al intentar hacer uso de estos servicios, la aplicación lanzará una ventana emergente (*AlertView*) preguntando por los permisos de acceso. Los sistemas de localización están centrados en la brújula, en el GPS y wifi.

15.1.8.10. Notificaciones Push

Las notificaciones push permiten a los usuarios estar avisados en cualquier momento, independientemente si la aplicación está funcionando en ese mismo instante. Esto es muy útil en aplicaciones que hagan uso de un calendario, para avisar de futuros eventos, o aplicaciones con algún tipo de recordatorio, etc. También se utiliza muy a menudo en juegos, sobre todo en juegos sociales. La implementación de este servicio por parte del desarrollador es algo compleja y requiere de un servidor propio, aunque se pueden

contratar servicios externos que realicen esa tarea de forma bastante económica. Al mismo tiempo que se activa la notificación push se muestra un número en la esquina superior derecha del ícono de la aplicación, a esto se le llama *badge* y es muy sencillo de implementar en nuestras aplicaciones.



Notificación push



Ejemplo badge en app

15.2. Personalización avanzada: celdas

La captura siguiente muestra la pantalla principal del cliente de Twitter para iPhone *Twinkle* y, aunque no lo parezca, es simplemente una tabla con las celdas totalmente personalizadas.



Ejemplo personalización Twinkle

La personalización de las celdas de una tabla es algo muy habitual en la programación de aplicaciones iOS, con ello podremos hacer que nuestras tablas se distingan del resto usadas comúnmente así como ofrecer un "toque" característico y concordante con nuestras aplicaciones. Pero, ¿cómo podemos hacer este tipo de celdas en nuestra aplicación iOS? Muy simple, a continuación veremos mediante un ejemplo paso a paso el diseño y programación de celdas personalizadas mediante el *Interface Builder* de XCode.

15.2.1. Creando el proyecto y las clases básicas

Al terminar este ejemplo tendremos una aplicación formada únicamente una vista de tabla *UITableView* con celdas personalizadas. Estas celdas tendrán una imagen en el lado izquierdo, un texto en negrita en la parte superior, un texto normal en la parte central y otro texto pequeño en la parte inferior. Las celdas tendrán un tamaño algo mayor al que viene por defecto y, para finalizar, la tabla tendrá estilo zebra, esto es que el fondo de las celdas se intercalará de color.

Comenzaremos creando el proyecto, para ello abrimos XCode y creamos un nuevo proyecto de tipo *Navigation-based Application* sólo para iPhone. Lo guardamos con el nombre *uisesion02-ejemplo1*. Con esto ya tenemos el esquema básico del proyecto: una vista de tabla contenida en un controlador de navegación. Ahora creamos una nueva vista

subclase de *Table View Cell*, para ello hacemos click en *New > New File*, seleccionamos *UIViewController Subclass* y *Subclass of UITableViewCell* **desmarcando** "With XIB for user interface" y "Targeted for iPad". La guardamos como *TableViewCell*.

15.2.2. Diseñando la celda desde Interface Builder

Una vez que hemos creado el proyecto y la clase de la celda ahora vamos a diseñarla, para ello creamos un nuevo archivo *New > New File* de tipo *User Interface* y seleccionamos la plantilla *Empty*. Guardamos el fichero con el nombre *TableViewCell*, lo abrimos y arrastramos desde la librería de objetos un *Table View Cell*:

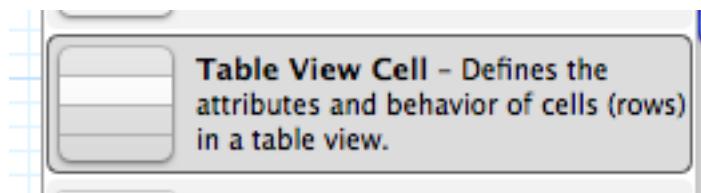
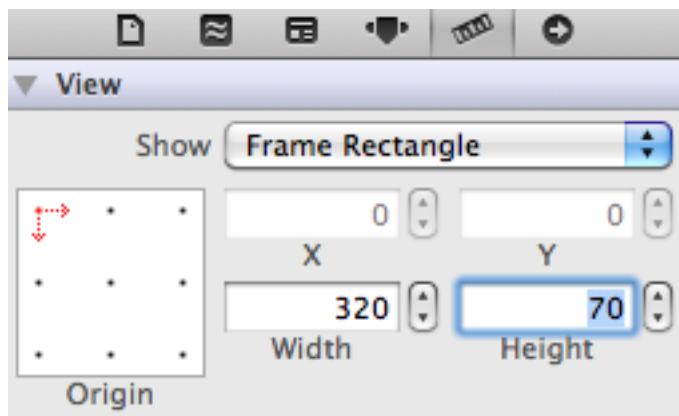


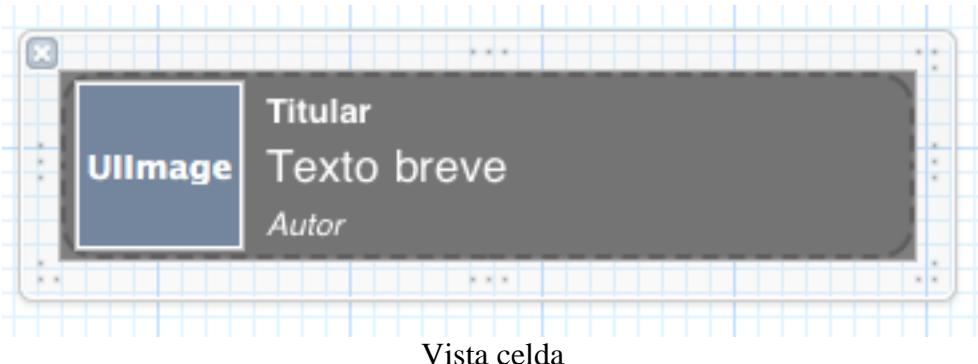
Table View Cell

Vamos a modificar su tamaño cambiándolo desde el *size inspector*, escribimos de altura: 70:



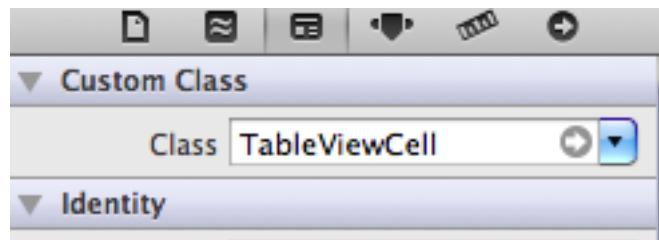
Ajustes de tamaño vista celda

Ahora queda arrastrar los objetos que queramos que aparezcan en la celda: 3 labels y una imagen por el momento. La celda queda de la siguiente manera en el *Interface builder*:



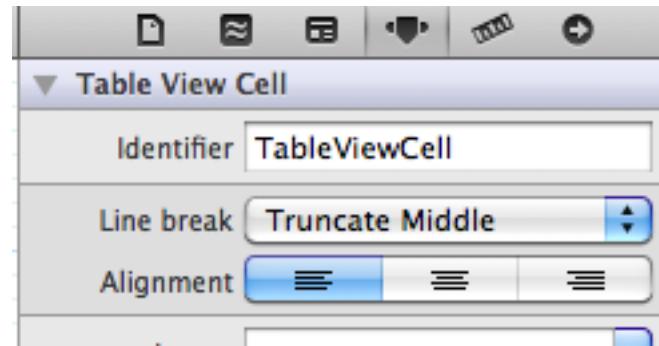
Vista celda

Ahora, en la pestaña de *Identity Inspector* escribimos la clase que corresponde a la vista, en nuestro caso será *TableViewCell*:



Asignamos la clase TableViewCell en Interface Builder

En la pestaña de *Attributes Inspector* escribimos un identificador para la celda: *TableViewCell*. Esto servirá más adelante para referenciar a ella desde el controlador de la tabla y así poder utilizar distintas celdas en la misma tabla si queremos:



Identificador en la celda

Con esto tenemos ya la vista de la celda creada. Ahora vamos a programar la clase.

15.2.3. Programando la celda

Dentro de la clase de la celda tendremos que añadir los atributos Outlet que hemos creado antes en la vista, para ello abrimos el fichero *TableViewCell.h* y escribimos lo siguiente:

```
#import <UIKit/UIKit.h>

@interface TableViewCell : UITableViewCell {
    UILabel *_labelTitulo;
    UILabel *_labelTexto;
    UILabel *_labelAutor;
    UIImageView *_imagen;
}

@property (nonatomic, retain) IBOutlet UILabel *labelTitulo;
@property (nonatomic, retain) IBOutlet UILabel *labelTexto;
@property (nonatomic, retain) IBOutlet UILabel *labelAutor;
@property (nonatomic, retain) IBOutlet UIImageView *imagen;

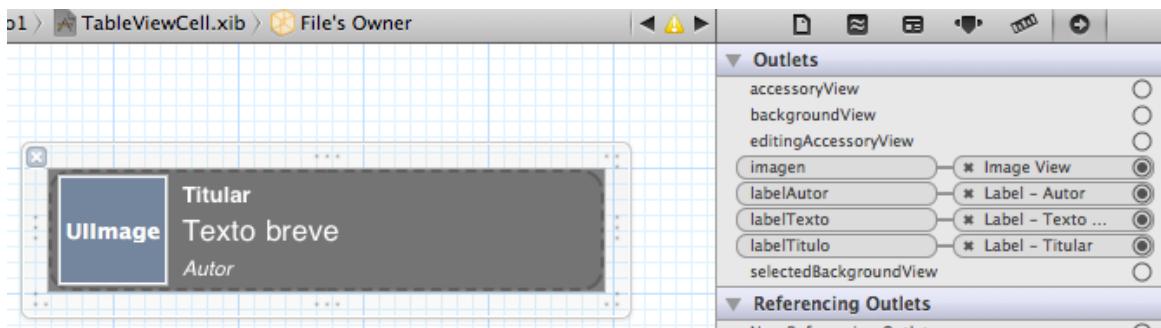
@end
```

En el fichero *TableViewCell.m* añadimos lo siguiente:

```
//Debajo de @implementation:

@synthesize labelTitulo=_labelTitulo;
@synthesize labelTexto=_labelTexto;
@synthesize labelAutor=_labelAutor;
@synthesize imagen=_imagen;
```

Ahora volvemos a la vista de la celda *TableViewCell.xib* y enlazamos los Outlets creados en la clase con los objetos de la vista:



Outlets de la vista celda

Una vez hecho esto abrimos de nuevo la clase controladora de la tabla *RootViewController.m* y modificamos los siguientes métodos:

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    return 70.0f;
}

// Customize the number of sections in the table view.
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}
```

```

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return 3; // para pruebas
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"TableViewCell";

    TableViewCell *cell = (TableViewCell*)[tableView
dequeueReusableCellWithIdentifier:
CellIdentifier];
    if (cell == nil) {
        // Cargamos el Nib de la celda con ese identificador
        NSArray *array = [[NSBundle mainBundle]
loadNibNamed:@"TableViewCell"
owner:self
options:nil];
        cell = [array objectAtIndex:0];
    }

    // Configure the cell.

    cell.labelTitulo.text = [NSString stringWithFormat:
@"Título noticia número %d", indexPath.row+1];
    cell.labelTexto.text = @"Texto de pruebas...";
    cell.labelAutor.text = @"Autor/es de la noticia";

    cell.imagen.image = [UIImage imageNamed:@"logo_mvl.png"];
    return cell;
}

```

La imagen *logo_mvl.png*la puedes descargar desde [aqui](#) y la debes de arrastrar al directorio de *Supporting files del proyecto*. Ahora ya podemos ejecutar la aplicación y nos debe de aparecer la tabla con las celdas que acabamos de programar.

Una vez que tenemos las celdas hechas vamos a aumentar un nivel más la personalización añadiendo un fondo determinado a las celdas pares y otro a las impares, para hacer un efecto "zebra". Para hacer esto necesitaremos dos imágenes más que las puedes descargar [aquí](#).

Abrimos el nib de la celda *TableViewCell.xib* y arrastramos un *ImageView* al fondo de la celda ocupando todo el espacio. En el fichero *TableViewCell.h* añadimos un *UIImageView* como Outlet y después lo enlazamos desde la vista, como hemos hecho con el resto de elementos.

Ahora en el fichero *RootViewController.m*, dentro del método *cellForRowAtIndexPath* añadimos lo siguiente justo antes de *return cell;*

```

if (indexPath.row%2){
    cell.fondo.image = [UIImage imageNamed:@"fondo_celdal.png"];
}

```

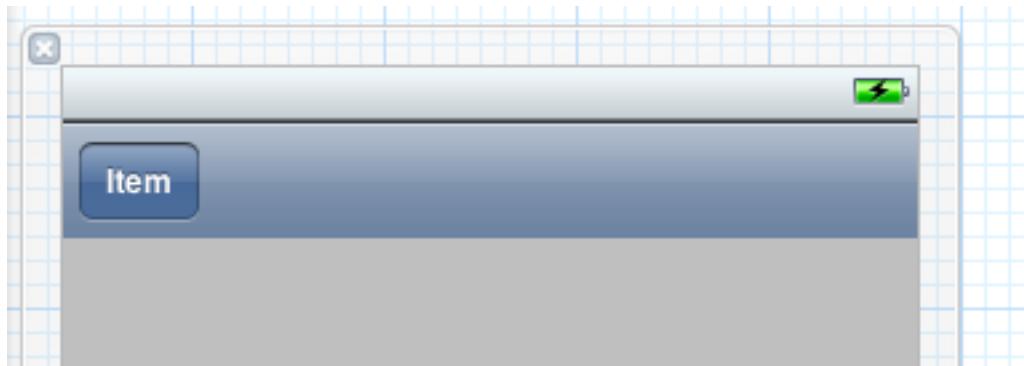
```
else {
    cell.fondo.image = [UIImage imageNamed:@"fondo_celda2.png"];
}
```

Después de modificar alguna propiedad de la tabla desde el *Interface Builder*, ejecutamos ahora el proyecto tendremos nuestra tabla con las celdas personalizadas.

15.3. Personalización de ToolBars

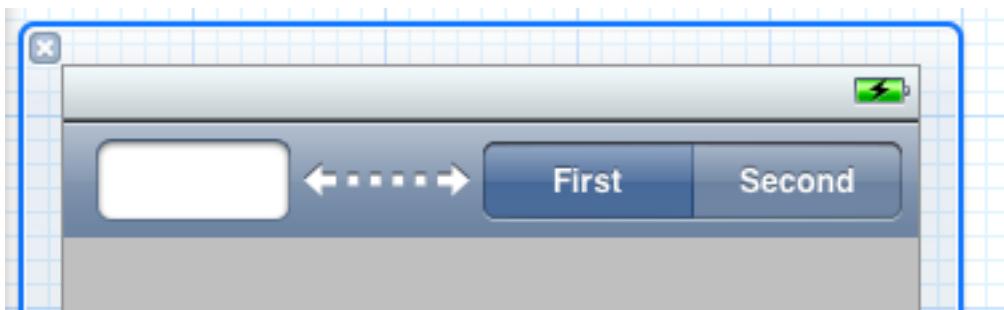
El *ToolBar*, al igual que la gran mayoría de componentes, se puede personalizar bastante mediante código para de esta forma conseguir las funcionalidades que deseemos en nuestras aplicaciones de iOS. En el siguiente ejemplo vamos a crear una vista *ToolBar* con un estilo determinado usando una imagen de fondo, estilos para los botones, añadiendo botones de distinto tipo, etc.

Comenzamos creando un nuevo proyecto de tipo *View-based application* con el nombre *uisesion02-ejemplo3*. Abrimos la vista *uisesion02_ejemplo3ViewController.xib* y arrastramos un objeto *ToolBar* a la vista, la situamos en la parte superior de ventana. También situamos un objeto *Label* en el centro de la vista.



ToolBar básico

Ahora vamos a situar los elementos que deseemos sobre el *ToolBar*, en nuestro caso arrastramos por el siguiente orden desde el listado de objetos de la columna de la derecha un *Text Field*, un *Flexible Space Bar* y un *Segmented Control*. El *ToolBar* quedará de la siguiente manera:



ToolBar con botones y campos de texto

Como podemos observar, la función del objeto *Flexible Space* no es más que añadir un espacio flexible entre dos objetos dentro de un *ToolBar*. Una vez que tenemos la barra con todos sus elementos vamos a definir los elementos dentro de la clase. El fichero *uisession02_ejemplo3ViewController.h* debe de quedar de la siguiente manera:

```
#import <UIKit/UIKit.h>

@interface uisession02_ejemplo3ViewController : UIViewController {
    UIToolbar *_toolBar;
    UITextField *_textField;
    UISegmentedControl *_segmentedControl;
    UILabel *_segmentLabel;
}

@property (nonatomic, retain) IBOutlet UIToolbar *toolBar;
@property (nonatomic, retain) IBOutlet UITextField *textField;
@property (nonatomic, retain) IBOutlet UISegmentedControl
*segmentedControl;
@property (nonatomic, retain) IBOutlet UILabel *segmentLabel;

@end
```

Y en el fichero *uisession02_ejemplo3ViewController.m*:

```
// Justo debajo del @implementation
@synthesize segmentedControl=_segmentedControl;
@synthesize textField=_textField;
@synthesize toolBar=_toolBar;
@synthesize segmentLabel=_segmentLabel;
```

Seguidamente tenemos que enlazar los *Outlets* dentro de la vista. En la vista nos situamos dentro del objeto *File's Owner* y arrastramos cada uno de sus Outlets hasta el elemento que corresponda. De esta forma ya tenemos "conectados" los elementos de la clase con los de la vista, ahora vamos a implementar la acción del *Segmented Control*, para ello escribimos la siguiente definición del método dentro de la clase *uisession02_ejemplo3ViewController.h* y lo implementamos en el *.m*:

```
//uisession02_ejemplo3ViewController.h
-(IBAction) segmentedControlIndexChanged;

//uisession02_ejemplo3ViewController.m
-(IBAction) segmentedControlIndexChanged{
    switch (self.segmentedControl.selectedSegmentIndex) {
        case 0:
            self.segmentLabel.text = @"Segmento 1
seleccionado.";
            break;
        case 1:
            self.segmentLabel.text = @"Segmento 2
seleccionado.";
```

```

        break;

    default:
        break;
    }
}

```

Para que el método se llame cuando se pulsa un botón del *Segmented Control* debemos enlazarlo dentro de la vista, para ello seleccionamos con la tecla cntrl pulsada el objeto *Segmented Control* y arrastramos hasta el objeto *File's Owner*. Ahí seleccionamos el método que hemos declarado justo antes: `segmentedControlIndexChanged`. En este punto ya podemos ejecutar el proyecto por primera vez y comprobar que al pulsar sobre uno de los botones del *Segmented Control* la etiqueta *Label* cambia.

Ya tenemos un objeto *ToolBar* con una personalización básica funcionando. Ahora vamos a personalizarlo un poco más: vamos a añadirle una imagen de fondo, a asignar un color de fondo al *Segmented Control* y a modificar el diseño del *TextField*. Para hacer todo esto debemos de modificar el método `viewDidLoad` de la clase `uisesion02_ejemplo3ViewController` y añadir el siguiente código:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Asignamos la imagen de fondo para el toolbar
    UIImageView *iv = [[UIImageView alloc] initWithImage:
    [UIImage imageNamed:@"fondo_madera.png"]];
    iv.frame = CGRectMake(0, 0, _toolBar.frame.size.width,
    _toolBar.frame.size.height);
    iv.autoresizingMask = UIViewAutoresizingFlexibleWidth;

    // Añadimos la imagen al toolbar. Distinguimos si estamos en iOS 4 o
    en iOS 5
    if([[UIDevice currentDevice] systemVersion] intValue] >= 5)
        [_toolBar insertSubview:iv atIndex:1]; // iOS5 atIndex:1
    else
        [_toolBar insertSubview:iv atIndex:0]; // iOS4 atIndex:0

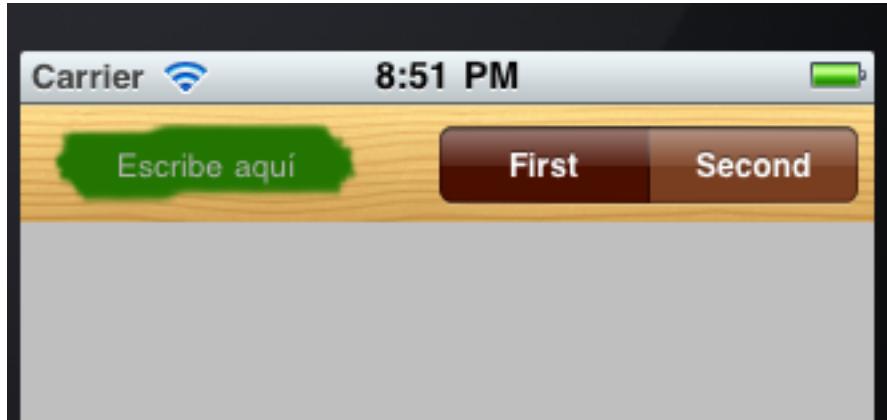
    // Añadimos color al Segmented Control
    [_segmentedControl setTintColor:[UIColor brownColor]];

    // Personalizamos el text field:
    _textField.textAlignment = NSTextAlignmentCenter; //centramos el texto
    _textField.textColor = [UIColor whiteColor]; //texto de color blanco
    _textField.borderStyle = UITextBorderStyleNone; //quitamos el borde
    del campo
    _textField.background = [UIImage imageNamed:@"fondo_textfield.png"];
    //fondo
    [_textField setPlaceholder:@"Escribe aquí"]; //texto inicial
}

```

Para que el código funcione debemos de descargarnos las imágenes desde [aquí](#) e insertarlas en la carpeta *Supporting Files* del proyecto. Una vez hecho esto ya podemos ejecutar el proyecto y veremos como ha cambiado. De esta forma tendremos nuestro componente *ToolBar* bastante personalizado, lo que le da un aspecto visual bastante más

agradable a la aplicación.



ToolBar personalizado

Por último comentar que el método que acabamos de implementar nos sirve tanto para personalizar el componente *TabBar* como el *Navigation Bar*.

16. Guías de estilo y personalizaciones avanzadas - Ejercicios

16.1. Personalizando las celdas de un Table View (1)

- a) Creamos un nuevo proyecto en XCode que llamaremos `CeldasPersonalizadas` el cual mostrará un listado de películas. El proyecto va a ser de tipo *Single View Application*, organización `es.ua.jtech` y prefijo `UA`. Seleccionamos `iPhone` como "target" y desmarcaremos todas las casillas adicionales, ya que no vamos a utilizar por el momento ninguna de esas características (Core Data, ARC, pruebas de unidad, storyboards).
- b) Abrimos la vista `UAVViewController.xib` y añadimos una vista de tabla `Table View` a la vista principal. La ajustamos para que ocupe toda la pantalla.
- c) Creamos el Outlet necesario dentro de la clase `UAVViewController` y lo relacionamos dentro de la vista. Modificamos la declaración de la clase para añadirle los protocolos delegados de `UITableViewDelegate`. Relacionamos los delegados de la vista de la tabla con la clase.
- d) Creamos la clase para la celda personalizada. A la clase la llamaremos `CeldaView`, será subclase de `UITableViewCell` y modificaremos su vista para añadirle los estilos que deseemos (labels, imágenes, botones, etc, etc). La celda debe de tener al menos dos labels y una imagen.
- e) Modificamos la controladora de la vista de la celda (`CeldaView`) añadiendo los Outlets necesarios para las labels, imágenes, etc que hayamos añadido a la vista de la celda y los relacionamos en la vista.
- f) Definimos los métodos delegados necesarios para gestionar la tabla, estos métodos serán los siguientes:

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    //devolvemos altura de la celda
}

// Customize the number of sections in the table view.
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    //devolvemos el numero de secciones (1)
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    //devolvemos el numero de filas por sección (10)
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath:
```

```
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"TableViewCell";
    CeldaView *cell = (CeldaView*)[tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        //Completamos con el código necesario para cargar la vista de la celda
    }
    // Configuramos la celda
    return cell;
}
```

g) Arrancamos la aplicación y comprobamos que todo funciona correctamente.

16.2. Personalizando las celdas de un Table View (2)

a) Seguimos con el ejercicio anterior. Vamos a "llenar" de datos las celdas de la tabla que hemos creado, para ello vamos a crear una clase a la que llamaremos Pelicula a cual contendrá un NSString que será el título, otro que será la sinopsis y una imagen (UIImage) que será la carátula.

b) Ahora crearemos un objeto de tipo NSMutableArray dentro de la clase UIViewController, lo inicializaremos y lo completaremos con al menos 5 películas que queramos. Cada elemento del array será de tipo Pelicula.

c) Una vez creado el array de películas vamos a mostrarlas en nuestra tabla, para ello deberemos de completar los métodos de la clase UITableView Delegate y completar los datos de las celdas correctamente.

d) Cuando hayamos terminado, comprobamos que la aplicación funciona según lo esperado.

16.3. Personalizando un Tool Bar

En este ejercicio deberemos de personalizar a nuestro gusto un elemento de tipo UIToolBar. Comenzamos creando un proyecto nuevo en XCode que llamaremos ToolBarPersonalizada. El proyecto será de tipo *Single View Application* para iPhone. Cuando tengamos el proyecto creado abrimos la vista principal y arrastramos un *Tab Bar View* desde el navegador del Interface Builder.

De forma programada tendremos que personalizar el Tab Bar añadiéndole al menos un botón, un *Segmented Control* y color o imagen de fondo.

