laSalle

UNIVERSITAT RAMON LLULL

Escola Tècnica Superior d'Enginyeria La Salle

Treball Final de Màster

Màster Universitari en Creació, Disseny i Enginyeria Multimèdia

iKlubbers: diseño e implementación de una aplicación iPhone (Vol. 2)

Alumne
David Bassols Espuña

Professor Ponent
Emiliano Labrador Ruiz de la Hermosa
Oscar García Pañella

ACTA DE L'EXAMEN
DEL TREBALL FI DE CARRERA

Reunit el Tribunal qualificador en el dia de la data, l'alumne

D. David Bassols Espuña

va exposar el seu Treball de Fi de Carrera, el qual va tractar sobre el tema següent:

iKlubbers: diseño e implementación de una aplicación iPhone (Vol. 2)

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Treball amb la qualificació de

Barcelona,

VOCAL DEL TRIBUNAL VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL

David Bassols

Abstract

Aquest projecte es basa en el desenvolupament d'una aplicació per iPhone[1] anomenada iKlubbers[2]. La plataforma iPhone és un dispositiu telefònic que va revolucionar el món de la telefonia mòbil, donant un nou concepte i forma en aquest mercat que s'estancava cada vegada més, permetent a tercers desenvolupar aplicacions per aquest dispositiu.

Amb aquesta aplicació es tindrà accés a tota la informació que es busca per gaudir de l'oci nocturn a l'abast de la mà. iKlubbers permet planejar les nits mostrant una completa llista dels clubs de la ciutat amb informació detallada per poder escollir millor, ja s'estigui a casa o viatjant, doncs iKlubbers cobreix tot el territori espanyol. A més, iKlubbers inclou llistes VIP per els millors locals, amb els que es tindrà accés preferent i gratuït entre molts altres avantatges.

Aquest projecte s'ha fet per l'empresa Mobivery[3], empresa desenvolupadora de gran quantitat d'aplicacions iPhone, la qual ha donat suport i ajuda en el desenvolupament d'iKlubbers.

Abstract

Este proyecto se basa en el desarrollo de una aplicación para iPhone[1] llamada iKlubbers[2]. La plataforma iPhone es un dispositivo telefónico que revolucionó el mundo de la telefonía móvil dando un nuevo concepto y forma en este mercado que se estancaba cada vez mas, permitiendo a terceros desarrollar aplicación para este dispositivo.

Con esta aplicación se tendrá acceso a toda la información que se busca para disfrutar del ocio nocturno al alcance de la mano. iKlubbers permite planear las noches mostrando una completa lista de los clubs de la ciudad con información detallada para poder escoger mejor, ya se esté en casa o viajando, pues iKlubbers cubre todo el territorio español. Además, iKlubbers incluye listas VIP para los mejores locales, con las que se tendrá acceso preferente y gratuito entre muchas otras ventajas.

Este proyecto se ha hecho para la empresa Mobivery[3], empresa desarrolladora de gran cantidad de aplicaciones iPhone, la cual ha dado soporte y ayuda en el desarrollo de iKlubbers.

Abstract

This project is based in the development of an iPhone[1] application called iKlubbers[2]. The iPhone platform is a telephonic device that revolutionized the world of the mobile telephony, giving a new concept and shape in this stagnated market, letting third parties develop applications for this device.

This application will give access to the whole information needed to enjoy the nightlife within the reach of you hand. iKlubbers allows planning the nights showing a complete list of the city's club, with the detailed information for a better choose, being at home or travelling, because iKlubbers covers the entire Spanish territory. Also, iKlubbers has VIP lists for the best locals, which gives preferred and free access among many other advantages.

This project has been done for the Mobivery[3] company, which develops a great amount of iPhone applications, and also gives support and help in the iKlubbers development.

Índice

Tabla de imágenes

1 Introducción

1.1 Resumen

Este proyecto se ha realizado como trabajo final del Master en Creación, Diseño y Engineria Multimedia (MCDEM)[4], que es un master multidisciplinar, donde se juntan estudiantes de diferentes perfiles para llevar a cabo un proyecto común.

Este proyecto proviene de una empresa externa a la universidad, que propone un trabajo al equipo, y este tiene que resolverlo en el tiempo establecido, aprendiendo las herramientas necesarias para llevarlo a cabo.

iKlubbers es un proyecto para iPhone, donde los usuarios podrán encontrar las discotecas cercanas mediante una lista o mapa, y mirar su información junto con las fotografías de los clubs y los comentarios de otros usuarios para poder escoger mejor.

La aplicación también cuenta con un buscador, una lista de destacados, unos favoritos y un historial, para poder acceder a la información del club que desees de la manera mas rápida y cómoda posible.

La característica principal de esta aplicación es la de poder apuntarse a listas VIP, donde se tendrá acceso preferente entre otras ventajas.

Esta memoria forma parte de un conjunto de memorias de los otros integrantes del grupo, y cada memoria explica una parte concreta del proyecto. Ésta está dividida en dos partes, una parte teórica y otra parte práctica En la parte teórica se explica el dispositivo iPhone, junto con las herramientas utilizadas para desarrollar la aplicación, así como el lenguaje de programación. En la parte práctica se explica como se ha estructurado la aplicación y como se ha llevado a cabo, junto con las decisiones tomadas y los problemas encontrados.

Me gustaría agradecer en primer lugar a los compañeros del equipo que he tenido pues han trabajado muy duro para llevar a cabo este proyecto, a los profesores Oscar García y Emiliano Ruiz por el apoyo dado y sus consejos para afrontar diferentes retos durante el desarrollo y a la empresa Mobivery, por darnos la oportunidad de poder trabajar en un proyecto de este calibre, aprendiendo de su metodología de trabajo y por ayudarnos en los problemas que surgían durante la implementación.

1.2 Marco

El mundo de la telefonía móvil ha cambiado radicalmente a partir de la salida del iPhone. Este dispositivo, a parte de ser un teléfono móvil con conexión a Internet permite a terceros desarrollar pequeños programas para él. Éstas aplicaciones le dan al iPhone más versatilidad y funcionalidades, a parte que permite hacer llegar tus aplicaciones a usuarios de todo el mundo.

Estas aplicaciones pueden hacer uso de Internet, el giroscopio[5] integrado, la cámara y todas las ventajas del dispositivo. Esto hace que puedan salir aplicaciones muy diversas y útiles.

La gente de hoy en día suele invertir su tiempo libre en discotecas y locales similares, pero el conocimiento de estos sitios es debido al boca a boca, poca gente busca en Internet. Aquellos que tienen iPhone, acceden a la información de qué clubs tienen cerca, pero la información a veces no es completa o no está, a parte que no se puede apuntar a listas VIP, punto importante a destacar. En la aplicación que proponemos se muestra información detallada de todos los clubs, i la disponibilidad de poder apuntarte a listas VIP, característica no disponible en las aplicaciones iPhone.

Para llevar a cabo tal tarea, se necesita de un ordenador de Apple[6], con el programa XCode[7] instalado. Este programa viene con el mismo ordenador y sirve para poder programar en muchos lenguajes[8], entre ellos el Objective C [9], que es el lenguaje usado para este sistema.

1.3 Estado del arte

Actualmente hay muchos terminales parecidos al iPhone, pero lo mas importante no es el teléfono sino el sistema operativo[10] que incorporan. Los sistemas operativos mas importantes son el iPhone OS para iPhone, y el Android[11], que es de código abierto, disponible para muchas otras plataformas. Estos sistemas se basan en tener una tienda virtual donde descargar aplicaciones con las que ampliar las capacidades de tu teléfono. La principal diferencia entre un iPhone y un móvil con Android, son su filosofía. Apple tiene una filosofía mas estricta en cuanto a las aplicaciones que se quieran subir para ser descargadas, con lo que Android da mas libertad, pero las aplicaciones subidas para iPhone, en consecuencia, suelen estar mas trabajadas en el detalle.

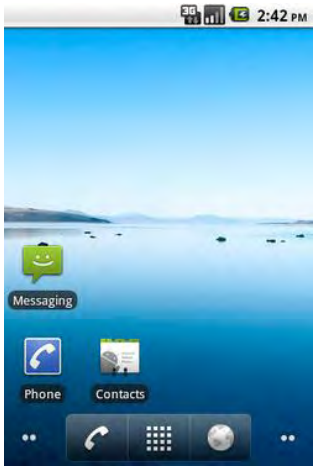

Ilustración 1 Sistema Android, por cortesía de wikipedia

Ilustración 2 iPhone, por cortesía de http://misioncumplida.wordpress.com/

Para apuntarse a una lista VIP o para ver los clubs nocturnos hay dos vías para hacerlo. Una es buscando en páginas Web los clubs que existen y su ubicación con una explicación del sitio. La mayoría de sitios Web permite al usuario ver un listado de los clubs de una ciudad ordenada alfabéticamente, u ordenada según otro criterio que puede ser relevancia o tipo de local. En ellas se pueden ver fotografías y una descripción del sitio para que el usuario sepa que tipo de local es junto con información de contacto.

Para entrar en una lista VIP, se tiene que ir a la página Web de la discoteca, o en algunas otras páginas Web, donde hay una breve lista de discotecas accesibles para tener lista VIP. El hecho es que no hay una

página Web común que tenga acceso a todas las listas VIP de todos los locales

En cuanto a aplicaciones de iPhone, se han encontrado diferentes aplicaciones que entre otras cosas buscan discotecas. Una de ellas y quizás la mas importante, bliquo[12] puede localizar restaurantes, bares cafeterías o discotecas. Ésta permite filtrar por tipo de discoteca que se quiera y muestra las discotecas por orden de cercanía, mostrando la valoración de la gente que ha valorado el sitio. También puede mostrar el detalle de cada club con una descripción y las fotografías del local. Esta información es publicada por los usuarios, y cualquier persona puede poner información en la aplicación como añadir mas locales.

Otras aplicaciones como el AroundMe[13], muestran locales y establecimientos de muchos tipos cercanos al usuario, pero con poca información y con una estructura de la aplicación mas bien caótica, y con una información escasa del sitio.

1.4 Descripción del problema

El estado actual es apuntarse en listas VIP a través de paginas Web o teléfono, y por lo que a aplicaciones iPhone respeta, sirven para localizar locales, no para entrar en listas VIP. Los sitios Web son poco generales, pues necesitas buscar activamente por Internet para tener la información de todos los locales, y acceder a la Web propia del local para acceder a su lista. Esto necesita de conexión a Internet y tiempo para buscar, que se tiene cuando se esta en casa, pero no cuando se esta cenando con amigos y se quiere salir de fiesta.

En cuanto a aplicaciones iPhone, solucionan el hecho de poder buscar, en cualquier momento del día o de la noche, el mejor sitio donde salir, pero se tiene una información mínima del contenido. Además, no puedes apuntarte a listas VIP para poder entrar gratis a las discotecas.

Así vemos que las webs tienen el problema que apuntarse es costoso, por el hecho de tardar mas tiempo, y las aplicaciones iPhone no dan todo el soporte que dan las webs.

1.5 Solución propuesta

La solución propuesta proviene en parte de la empresa Mobivery, y se trata de una aplicación que junte la información de las paginas Web, junto con la posibilidad de poder apuntarse a las listas y el dinamismo de las aplicaciones iPhone por el hecho de que se pueden ejecutar en cualquier momento y lugar.

Esta aplicación llamada iKlubbers, tiene como cliente los clubs, que tendrán que pagar unas tarifas para poder aparecer en la aplicación. Esto se hace por que podrán gestionar su propia información así como añadir fotografías del local, y poder aparecer en un apartado de destacados.

La aplicación muestra las discotecas cercanas al usuario, ahorrando la necesidad de buscar, además, incluye listas VIP, mientras que las otras aplicaciones no lo permiten, donde se podrá apuntar a una lista de un club, con pocas acciones.

De esta forma hemos conseguido una aplicación con una gran cantidad de información, fácil de usar y dinámica, para que se pueda llevar en todos los sitios.

2 Fundamentos teóricos

En este apartado se explicaran las herramientas que se han utilizado para el desarrollo de la aplicación, una explicación de que es el iPhone así como el lenguaje de programación y las clases mas importantes.

2.1 iPhone

El **iPhone**, de la compañía Apple Inc., es un teléfono inteligente[14] multimedia con conexión a Internet[15], pantalla táctil[16] (con tecnología multitáctil[17]) y una interfaz de hardware[18] minimalista[19].

Ya que carece de un teclado físico, se muestra uno virtual en la pantalla. El iPhone 3GS dispone de una cámara de fotos de 3 megapíxeles[20] y un reproductor de música (equivalente al del iPod) además de software[21] para enviar y recibir mensajes de texto y mensajes de voz. También ofrece servicios de Internet como leer correo electrónico, cargar páginas Web y conectividad por Wi-Fi.

Ilustración 3 Iphone, cedida por apple.com

2.1.1 Pantalla e interfaz

Casi todas las órdenes se dan con la botonera, que es capaz de entender gestos complejos usando botones. Las técnicas de interacción del iPhone hacen que el usuario sea capaz de mover el contenido arriba o abajo simplemente con tocar un botón cualquiera. Por ejemplo, para aumentar o reducir el zoom de imágenes y páginas Web hay que poner opción con el botón izquierdo y poner zoom en la pantalla y separarlos o acercarlos. De forma similar, el movimiento del botón hacia arriba o hacia abajo de la pantalla imita la rueda de un ratón de PC. Ya que la fricción activa este movimiento, la página decelerará hasta detenerse si no se mantiene el

contacto con la pantalla. Así, la interfaz simula la física de un objeto real en 3D. Hay otros efectos visuales, como deslizar subsecciones de derecha a izquierda, desplazar de arriba abajo los menús del sistema (como por ejemplo, la sección de 'favoritos'), y los menús y widgets[22] a los que se puede dar la vuelta y que muestran opciones de configuración por detrás.

El visualizador responde a tres sensores[23]. Un sensor de proximidad apaga el visualizador y la pantalla táctil cuando se pone el iPhone cerca de la cara para ahorrar batería y prevenir que algún botón se pulse accidentalmente al contacto con la piel de la cara y la oreja. Un sensor de luz ambiental ajusta el brillo del visualizador, lo que además de proteger la vista ahorra también batería. Un acelerómetro[24] de tres ejes detecta la orientación del teléfono y cambia la pantalla según esté colocado. Se puede ver fotos, páginas Web, y portadas de discos en horizontal y en vertical, desde todos los sentidos, pero los videos sólo pueden visualizarse en horizontal y en un único sentido, con el botón de inicio a la derecha.

Una actualización de software permitía a la primera generación de iPhones usar antenas de telefonía móvil y puntos de acceso Wi-Fi para localizar su posición a pesar de carecer de GPS. El iPhone 3G incluye AGPS pero también necesita antenas de telefonía móvil y Wi-Fi para que funcione.

Un simple botón de inicio situado debajo de la pantalla nos lleva al menú principal. Las otras selecciones se hacen con la pantalla táctil. El iPhone se visualiza a toda pantalla, con submenús específicos arriba o abajo de cada página según el contexto, que se agrandan o encogen dependiendo de la orientación de la pantalla. Las páginas más importantes tienen el botón 'atrás' para volver al menú principal.

El iPhone tiene tres botones a los lados: apagar/encender el teléfono, subir/bajar el volumen y botón de silencio. Están hechos de plástico en el iPhone original y de metal para el iPhone 3G. Todas las demás operaciones multimedia y del teléfono se realizan con la pantalla táctil. El iPhone 3G

tiene por detrás una carcasa de plástico negro para incrementar la fuerza de las señales GSM.8 El iPhone 3GS además cuenta con la característica de poder optar por una carcasa de color blanco, disponible para todos sus modelos.

2.1.2 Sistema operativo

El iPhone OS es el sistema operativo que utiliza el iPhone y el iPod touch. Está basado en una variante del Mach kernel que se encuentra en Mac OS X[25]. El iPhone OS incluye el componente de software "Core Animation" de Mac OS X v10.5 que, junto con el PowerVR[26] MBX el hardware de 3D, es responsable de las animaciones usadas en el interfaz de usuario. IPhone OS tiene 4 capas de abstracción: la capa del núcleo del sistema operativo, la capa de Servicios Principales, la capa de Medios de comunicación y la capa de Cocoa Touch. El sistema operativo ocupa bastante menos de medio gigabyte del total del dispositivo, de 8 GB o de 16 GB.12 Esto se realizó para poder soportar futuras aplicaciones de Apple, así como aplicaciones de terceros publicadas en la iTunes Store[27] o la App Store[28].

Este sistema operativo no tenía un nombre oficial hasta que salió la primera versión beta del iPhone SDK, el 6 de marzo de 2008. Antes de esto, Apple declaró, que *"el iPhone controla un sistema operativo X"*, una referencia al padre de los sistemas operativos de los iPhone, el Mac OS X.

Como un iPod, el iPhone se maneja con la versión 7.3 o superior de iTunes, el cual es compatible con Mac OS X versión 10.4.10, y con Windows XP o Windows Vista de 32 bits. A partir de la versión 7.6 de iTunes se incluyeron las versiones de 64 bits del Windows XP y del Vista, y se han facilitado parches para versiones anteriores de Windows.

Las aplicaciones del iPhone no se pueden copiar directamente de Mac OS X ya que tienen que ser escritas y compiladas específicamente para el

iPhone. Además, el navegador Safari[29] soporta aplicaciones Web escritas con AJAX[30].

2.2 Objective-C

2.2.1 Introducción

Objective-C es un lenguaje de programación orientado a objetos creado como un superconjunto de C pero que implementa un modelo de objetos parecido al de SmallTalk[31]. Originalmente fue creado por Brad Cox[32] y la corporación StepStone[33] en 1980. En 1988 fue adoptado como lenguaje de programación de NEXTSTEP[34] y en 1992 fue liberado bajo licencia GPL para el compilador GCC[35]. Actualmente se usa como lenguaje principal de programación en Mac OS X y GNUstep.

2.2.2 Sintaxis

Objective-C es una muy fina capa por encima de C, y además es un estricto superconjunto de C. Esto es, es posible compilar cualquier programa escrito en C con un compilador de Objective-C, y también puede incluir libremente código en C dentro de una clase de Objective-C. La sintaxis de objetos de Objective-C deriva de Smalltalk. Toda la sintaxis para las operaciones no orientadas a objetos (incluyendo variables primitivas, pre-procesamiento, expresiones, declaración de funciones y llamadas a funciones) son idénticas a las de C, mientras que la sintaxis para las características orientadas a objetos es una implementación similar a la mensajería de Smalltalk.

2.2.2.1 Mensajes

El modelo de programación orientada a objetos de Objective-C se basa en enviar mensajes a instancias de objetos. Esto es diferente al modelo de programación al estilo de Simula, utilizado por C++ y ésta distinción es semánticamente importante. En Objective-C uno no *llama a un método*;

uno *envía un mensaje*, y la diferencia entre ambos conceptos radica en cómo el código referido por el nombre del mensaje o método es ejecutado. En un lenguaje al estilo Simula, el nombre del método es en la mayoría de los casos atado a una sección de código en la clase objetivo por el compilador, pero en Smalltalk y Objective-C, el mensaje sigue siendo simplemente un nombre, y es resuelto en tiempo de ejecución: el objeto receptor tiene la tarea de interpretar por sí mismo el mensaje. Una consecuencia de esto es que el mensaje del sistema que pasa no tiene chequeo de tipo: el objeto al cual es dirigido el mensaje (conocido como *receptor*) no está inherentemente garantizado a responder a un mensaje, y si no lo hace, simplemente lo ignora y retorna un puntero nulo.

Enviar el mensaje `method` al objeto apuntado por el puntero `obj` requeriría el siguiente código en C++:

```
obj->method(parameter);
```

mientras que en Objective-C se escribiría como sigue:

```
[obj method:parameter];
```

Ambos estilos de programación poseen sus fortalezas y debilidades. La POO[36] al estilo Simula permite herencia múltiple[37] y rápida ejecución utilizando ligadura en tiempo de compilación siempre que sea posible, pero no soporta ligadura dinámica por defecto. Esto fuerza a que todos los métodos posean su correspondiente implementación, al menos que sean virtuales (una implementación es requerida aún por el método para ser llamada). La POO al estilo Smalltalk permite que los mensajes no posean implementación - por ejemplo, toda una colección de objetos pueden enviar un mensaje sin temor a producir errores en tiempo de ejecución. El envío de mensajes tampoco requiere que un objeto sea definido en tiempo de compilación.

Sin embargo, se debe notar que debido a la sobrecarga de la interpretación de los mensajes, un mensaje en Objective-C toma, en el mejor de los casos, tres veces más tiempo que una llamada a un método virtual en C++.

2.2.2.2 Interfaces e implementaciones

Objective-C requiere que la interfaz e implementación de una clase estén en bloques de código separados. Por convención, la interfaz es puesta en un archivo cabecera y la implementación en un archivo de código; los archivos cabecera, que normalmente poseen el sufijo .h, son similares a los archivos cabeceras de C; los archivos de implementación (método), que normalmente poseen el sufijo .m, pueden ser muy similares a los archivos de código de C.

2.2.2.2.1 Interfaz

La interfaz de la clase es usualmente definida en el archivo cabecera. Una convención común consiste en nombrar al archivo cabecera con el mismo nombre de la clase. La interfaz para la clase `Clase` debería, así, ser encontrada en el archivo `Clase.h`.

La declaración de la interfaz de la forma:

```
@interface classname : superclassname {
    // instance variables
}
+classMethod1;
+(return_type)classMethod2;
+(return_type)classMethod3:(param1_type)parameter_varName;

-(return_type)instanceMethod1:(param1_type)param1_varName
:(param2_type)param2_varName;
-
(return_type)instanceMethod2WithParameter:(param1_type)param1_va
rName andOtherParameter:(param2_type)param2_varName;
```

```
@end
```

Los signos *más* denotan métodos de clase, los signos *menos* denotan métodos de instancia. Los métodos de clase no tienen acceso a las variables de la instancia.

Si usted viene de C++, el código anterior es equivalente a algo como esto:

```
class classname : superclassname {
  public:
    // instance variables

    // Class (static) functions
    static void* classMethod1();
    static return_type classMethod2();
    static          return_type          classMethod3(param1_type
parameter_varName);

    // Instance (member) functions
    return_type    instanceMethod1(param1_type    param1_varName,
param2_type param2_varName);
   return_type          instanceMethod2WithParameter(param1_type
param1_varName, param2_type param2_varName = default);
};
```

Note que instanceMethod2WithParameter demuestra la capacidad de nombrado de parámetro de Objective-C para la cual no existe equivalente directo en C/C++.

Los tipos de retorno pueden ser cualquier tipo estándar de C, un puntero a un objeto genérico de Objective-C, o un puntero a un tipo específico así como NSArray *, NSImage *, o NSString *. El tipo de retorno por defecto es el tipo genérico `id` de Objective-C.

Los argumentos de los métodos comienzan con dos puntos seguidos por el tipo de argumento esperado en los paréntesis seguido por el nombre del

argumento. En algunos casos (por ej. cuando se escriben APIs[38] de sistema) es útil agregar un texto descriptivo antes de cada parámetro.

```
-(void) setRangeStart:(int)start End:(int)end;
-(void)          importDocumentWithName:(NSString          *)name
withSpecifiedPreferences:(Preferences                      *)prefs
beforePage:(int)insertPage;
```

2.2.2.2.2 Implementación

La interfaz únicamente declara la interfaz de la clase y no los métodos en sí; el código real es escrito en la implementación. Los archivos de implementación (métodos) normalmente poseen la extensión *.m*.

```
@implementation classname
+classMethod {
    // implementation
}
-instanceMethod {
    // implementation
}
@end
```

Los métodos son escritos con sus declaraciones de interfaz. Comparando Objective-C y C:

```
-(int)method:(int)i {
    return [self square_root: i];
}
int function(int i) {
    return square_root(i);
}
```

La sintaxis admite pseudo-nombrado de argumentos.

```
-(int)changeColorToRed:(float)red green:(float)green
blue:(float)blue [myColor changeColorToRed:5.0 green:2.0
blue:6.0];
```

La representación interna de éste método varía entre diferentes implementaciones de Objective-C. Si myColor es de la clase `Color`, internamente, la instancia del método -changeColorToRed:green:blue: podría ser etiquetada como _i_Color_changeColorToRed_green_blue. La `i` hace referencia a una instancia de método, acompañado por los nombres de la clase y el método, y los dos puntos son reemplazados por guiones bajos. Como el orden de los parámetros es parte del nombre del método, éste no puede ser cambiado para adaptarse al estilo de codificación.

De todos modos, los nombres internos de las funciones son raramente utilizadas de manera directa, y generalmente los mensajes son convertidos a llamadas de funciones definidas en la librería en tiempo de ejecución de Objective-C – el método que será llamado no es necesariamente conocido en tiempo de vinculación: la clase del receptor (el objeto que envió el mensaje) no necesita conocerlo hasta el tiempo de ejecución.

2.2.2.2.3 Instanciación

Una vez que una clase es escrita en Objective-C, puede ser instanciada. Esto se lleva a cabo primeramente alojando la memoria para el nuevo objeto y luego inicializándolo. Un objeto no es completamente funcional hasta que ambos pasos sean completados. Esos pasos típicamente se logran con una simple línea de código:

```
MyObject * o = [[MyObject alloc] init];
```

La llamada a alloc aloja la memoria suficiente para mantener todas las variables de instancia para un objeto, y la llamada a init puede ser

anulada para establecer las variables de instancia con valores específicos al momento de su creación. El método init es escrito a menudo de la siguiente manera:

```
-(id) init {
    self = [super init];
    if (self) {
        ivar1 = value1;
        ivar2 = value2;
        .
        .
        .
    }
    return self;
}
```

2.3 XCode

XCode es un conjunto de herramientas para el desarrollo de software en Mac OS X, desarrollado por Apple. La principal aplicación del conjunto es la "integrated development environment" (IDE)[39], también llamado Xcode. El conjunto Xcode incluye también la mayor parte de la documentación para el desarrollo, y el "interface Builder[40]", una aplicación usada para construir interfaces graficas.

2.3.1 Xcode (IDE)

Xcode es una IDE completa, llena de características entorno a un flujo de trabajo suave que integra edición de código fuente, con los pasos de compilación[41] a través de una experiencia de depuración gráfica; todo sin perder la vista de tu código fuente.

La IDE hace mas que estas tareas tradicionales. Con el advenimiento del iPhone SDK[42], Xcode puede manejar todos los dispositivos de testeo, empaquetando automáticamente aplicaciones iPhone con los certificados correctos, e instalando aplicaciones en el mismo iPhone. El depurador[43] remoto conecta con el dispositivo en tiempo real, controlando los breakpoints[44] mientras la aplicación está en el dispositivo.

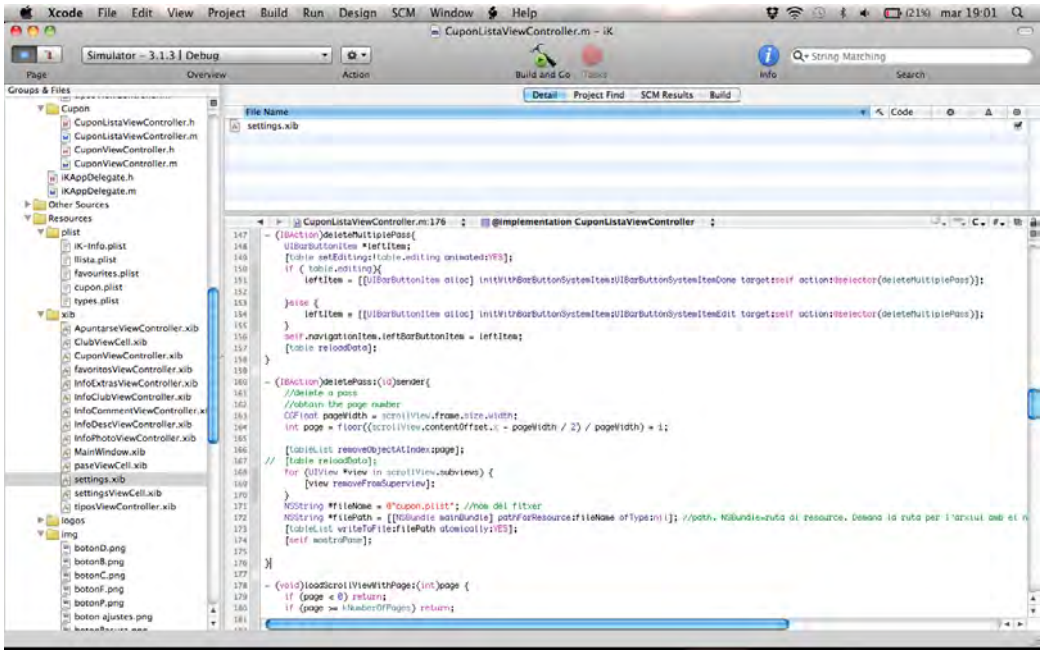

Ilustración 4 XCode

2.3.2 Interface Builder

Interface Builder es un editor grafico fácil de usar para diseñar cada aspecto de la interfaz gráfica de tu aplicación iPhone o Mac. Interface Builder guarda tu diseño en uno o mas archivos, como un conjunto de objetos de interfaz y sus relaciones. Los cambios que realices en la interfaz, se sincronizan automáticamente con el Xcode.

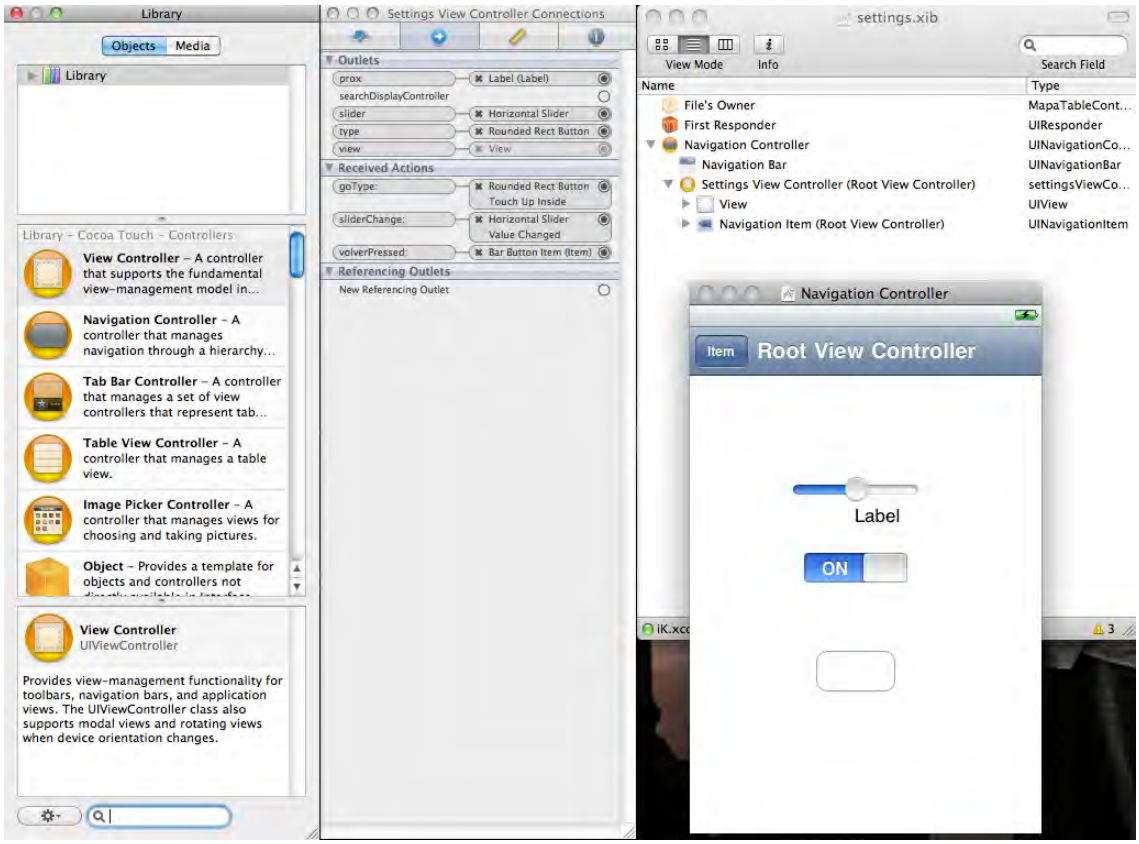

Ilustración 5 Interface Builder

2.3.3 iPhone Simulator

El iPhone Simulator[45] ejecuta tu aplicación de la misma forma como si fuera tu dispositivo iPhone. Como es rápido de ejecutar y depurar, el simulador se convierte en el mejor método de testeo, y comprobar que tu interfaz funciona de la forma que tu quieres, tus conexiones a Internet son correctas y que las vistas cambian correctamente cuando el iPhone gira. También se pueden simular gestos de pulsación con el ratón.

Ilustración 6 iPhone Simulator

3 Parte práctica

El proyecto iKlubbers se basa en el desarrollo de una aplicación para iPhone para poder entrar gratis en los clubs. Entrar en los clubs gratuitamente es lo mas importante para el usuario, pero para ello, tiene que ver la información de cada club. Para ello se han definido 5 rutas diferentes para llegar a la información de cada club, las cinco formas mas comunes con las que un usuario pueda llegar a la información del club. Se definieron 5 rutas pues Apple define en el iPhone la TabBar[46] i la NavBar[46]. La TabBar es la barra inferior con botones que sirve para ver las diferentes funcionalidades de la aplicación, mientras que la NavBar es la barra superior que sirve para ver la navegación dentro de una misma funcionalidad.

El iPhone solo permite ver 5 botones en la TabBar, porque si fueran mas pequeños seria difícil seleccionar uno u otro, pues serian demasiado pequeños. Si se quieren poner mas botones en la TabBar, se puede sin problema, pero el iPhone, automáticamente, cambia el quinto botón por uno de mas, i al seleccionar ese, se ve en pantalla los otros botones que no se podían ver en una lista, y así poder configurarlo.

Cada una de estas 5 funcionalidades se han desarrollado por separado y después se enlazaron con cada botón de la TabBar. Estas 5 funcionalidades están descritas a continuación con una explicación técnica y teórica.

Como en todas las diferentes funcionalidades hay un diseño común, se explicará esta parte de la integración del diseño aquí para no ponerlo de forma general.

La NavigationBar, que es la barra de navegación superior, tiene una imagen de fondo mas un título.

```
self.navigationItem.title = @"Clubs";
```

Con esta línea accedemos al título de la navigationBar, y le establecemos el título que queremos en cada pantalla.

```
@implementation UINavigationBar (BackGround)
-(void)drawRect:(CGRect)rect{
     UIImage *navImage = [UIImage imageNamed:@"nav bar.png"];
     [navImage drawInRect:CGRectMake(0,0,320, 44)];
}
@end
```

Para cambiar el aspecto de la navigationBar, la única forma para hacerlo es reescribiendo el método drawRect de la clase UINavigationBar. Éste código hace esto mismo, primer indicamos que vamos a implementar código del UINavigationBar, y luego ponemos las funciones que vamos a implementar, que en este caso es la drawRect. Dentro de la función, cargamos una imagen, y luego indicamos que la queremos pintar en las dimensiones indicadas. Éste método sirve para toda la UINavigationBar de toda la aplicación, así que no se tiene que volver a definir.

El siguiente punto a completar son los botones de la TabBar. Ahí, cada botón tiene una imagen y un texto especificando qué es. Desde el interface Builder se ponen las imágenes y textos que la diseñadora especificó en los campos imagen y texto. De esta forma tan sencilla se ha integrado el diseño de los botones, ya que además, cuando uno de los botones se pulsa, queda con un brillo azul en todo aquello que es blanco, hecho que el programa hace solo

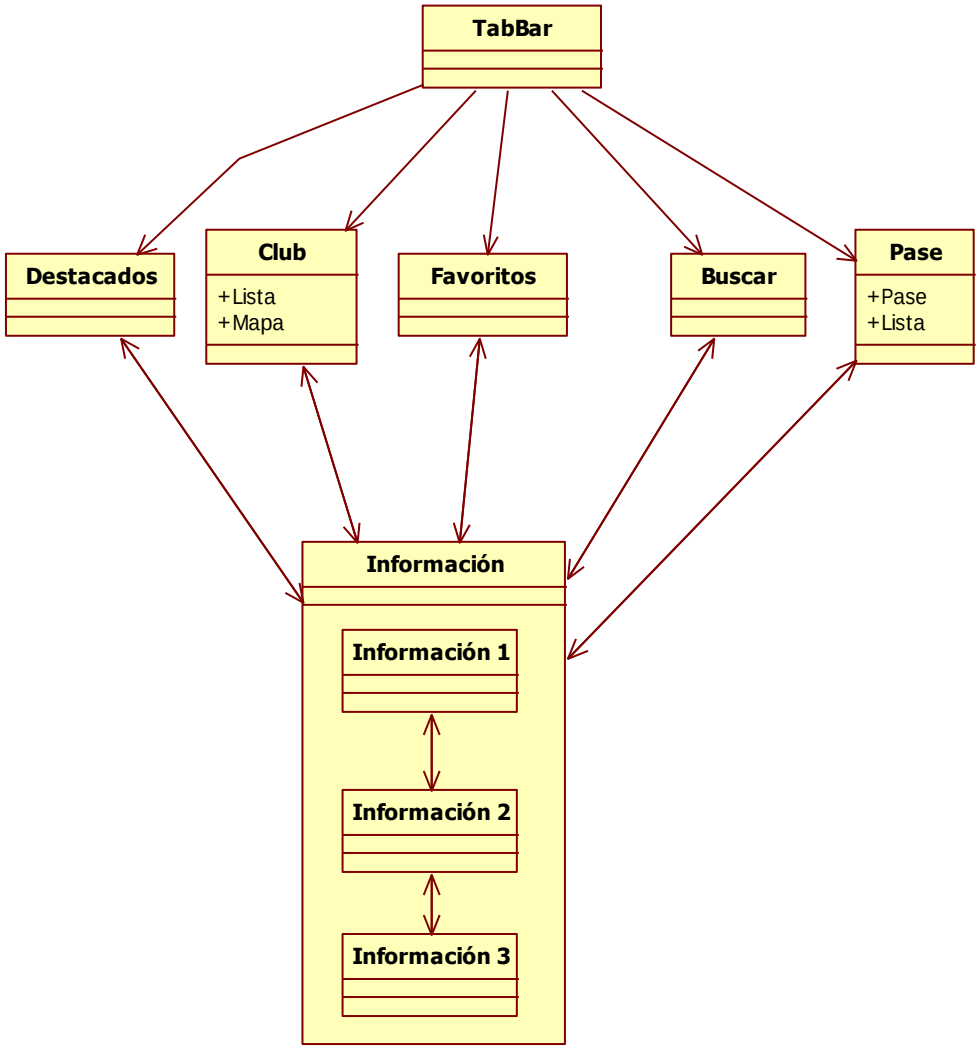

Ilustración 7 Diagrama aplicación

3.1 Destacados

Esta funcionalidad permite ver los pósters de los mejores clubs que tiene la aplicación mediante una animación rotatoria horizontal. En este apartado solo aparecen los mejores pósters. Tal como se especifica en la memoria de Marta Cortiñas, habrá 4 tipos de clubs. Según la cuota que paguen los clubs, se mostrará más información en la aplicación o menos, y los que paguen la mejor tarifa, tendrán el privilegio de poder poner sus pósters en esta funcionalidad.

Esta funcionalidad esta pensada para los clubs, pues es una forma de publicitarse. Tiene una animación atractiva, que llama al usuario a utilizarla para entretenerse, y gusta utilizarla. Esta es una buena forma para el club de plubicitarse, ya que mientras el usuario "juega" con la aplicación, ve los pósters de los mejores clubs, y puede decidirse por uno si le interesa.

Como los clubs son la fuente de ingresos de la aplicación, se ha decidido que esta se la primera pues, si no fuera así, los usuarios raramente utilizarían la funcionalidad de destacados, y no seria una buena publicidad para los clubs.

Para una explicación mas detallada y extensa, consultar la memoria de Antonio J. Gómez iKlubbers: Diseño e implementación de una aplicación iPhone - iPhone Vol. I

.

Ilustración 8 Destacados, cortesía de club Oshum

3.2 Clubs

En esta funcionalidad, se muestran aquellos clubs que se encuentran cercanos al usuario. A través de unos ajustes se podrá definir la distancia donde buscar clubs, si ver solo a los clubs donde se pueda entrar gratis, y el tipo de música que se prefiere.

Para ayudar al usuario, se han propuesto dos formas diferentes de mostrar la misma información, una mediante el mapa, y otra mediante una tabla.

Mediante la tabla, se muestran los clubs ordenados por proximidad. En esta tabla se puede ver el logotipo, el nombre, la valoración global, si dispone de lista para apuntarse y la distancia que te separa del club en metros. Esta lista también se puede ordenar por valoración, ver las mas valoradas al principio. Aunque se ordene por valoradas o por distancia, se muestran siempre las cercanas al usuario, solo cambia la ordenación de las mismas.

Si se cambia la visualización a mapa, se ve el mapa donde esta el usuario, con unas chinchetas con el logotipo de iKlubbers en la ubicación de cada club cercano al usuario. Cuando se pulsa una de estas chinchetas se ve el logotipo del club con su nombre. Además, hay un botón para poder ver la información completa del club.

Para poder personalizar toda esta información, se dispone de unos ajustes. En estos ajustes se puede definir la distancia donde buscar clubs, un botón para definir si quiere ver solo los clubs con pase gratis o todos, y finalmente una tabla donde definir los tipos de música que le gustan y filtrar por ellos.

Para una explicación mas detallada y extensa, consultar la memoria de Antonio J. Gómez, iKlubbers: Diseño e implementación de una aplicación iPhone - iPhone Vol. II

Ilustración 9 Clubs (lista y mapa)

3.3 Favoritos

Cuando se añade un club a la lista de Favoritos, aparece en este apartado. En este sitio, el usuario puede personalizar que clubs quiere tener y en que orden, de esta forma, no tendrá que buscarlos si son clubs que busca frecuentemente.

En el primer momento, para hacer las pruebas, se creó una lista de propiedades llamado plist. Este fichero tiene una estructura de XML específica que utiliza Apple para definir propiedades en las aplicaciones. En este fichero se añadieron varios clubs para comprobar que funcionaba.

Como la información de los clubs esta guardada en ficheros, accederemos a ellos y guardaremos dicha información en una variable que llamaremos data, y que es del tipo NSMutableArray. Este tipo de variable es un array, es decir, una colección de objetos listados uno detrás de otro, pero este tiene la ventaja de que incorpora muchos métodos para la edición de este mismo. Los otros arrays están mas cerrados, y no dejan modificarlos mucho o nada.

```
NSString *fílenme= @"favourites.plist";
NSString *filePath = [[NSBundle mainBundle] pathForResource:
fílenme ofType:nil];
```

Primero definimos el nombre del fichero, y se lo pasamos a la función pathForResource, que busca donde esta el fichero con el nombre que le dimos. De esta forma tenemos la ruta para encontrar el fichero y poder leerlo de manera fácil.

```
data = [[NSMutableArray alloc] initWithContentsOfFile:[self
dataFilePath]];
```

De esta forma, se inicializa nuestra variable data con la información del fichero con la ruta que se ha buscado antes. Ahora ya tenemos todos los clubs cargados en la memoria y listos para ser leídos.

```
[mainTableView reloadData];
```

Una vez hemos leído toda la información y la tenemos disponible, implementamos los métodos propios de una tabla.

```
-    (NSInteger)    tableView:    (UITableView    *)    tableView
numberOfRowsInSection:(NSInteger)section {
     return [data count];
}
```

Éste método define cuantas filas tendrá la tabla, es decir, las dimensiones en numero de celdas. Para saberlo, le damos el numero de objetos dentro nuestro array que contiene los clubs.

```
-    (UITableViewCell    *)tableView:(UITableView    *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
```

Con este método, se define el aspecto de cada celda. El método te da la referencia a la tabla y el número de celda en el que estamos.

```
static NSString *CellIdentifier = @"Cell";

UITableViewCell    *cell    =    (UITableViewCell    *)[tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
```

Primero de todo, creamos una variable celda, y la inicializamos. El método dequeueReusableCellWithIdentifier se tiene que llamar siempre, pues devuelve una celda con el nombre definido como CellIdentifier. Si la celda ya esta creada, te la devuelve, y no es necesario volver a crearla.

```
if(cell == nil){
     cell  =  [[[ClubViewCell  alloc]  initWithFrame:CGRectZero
reuseIdentifier:CellIdentifier]autorelease];
     cell.accessoryType                                          =
UITableViewCellAccessoryDisclosureIndicator;
}
```

Si la celda es null, es decir, no existe, la creamos, dándole las midas oportunas, y el mismo identificador, pues todas las celdas de una misma tabla, tienen que tener el mismo identificador.

También establecemos el AccessoryType. Esta propiedad es un indicador de control, y hay muchos tipos. Lo mas común es que indiquen si la celda de la tabla tiene mas contenido, si al pulsar en ella se va a mostrar una nueva ventana con mas información o no. Otras funcionalidades, por ejemplo, es poner checkmarks, es decir, poner una señal de si se ha seleccionado esa celda.

```
NSDictionary *inf=[[data objectAtIndex:indexPath.row]retain];
cell.imageView.image = [UIImage imageNamed:[inf
objectForKey:@"photo"]];
cell.text = [inf objectForKey:@”name”];
[inf release];
return cell;
```

Una vez hemos creado nuestra celda, tenemos que rellenarla con contenido. En este caso, hemos cogido un elemento del array data, el que corresponde al numero de celda. Una vez hecho esto, como la información es un diccionario, cogemos los valores de nombre photo y name guardados y los asignamos a la imagen i al texto de la celda. Por defecto, las celdas de una tabla tienen un espacio reservado para una imagen, situado a la izquierda del todo y también un texto. En el ejemplo superior se puede ver como se accede a dichos elementos de la celda. Para crear una celda con un diseño propio, se especificará en el apartado 3.7 CustomCell. Finalmente, se devuelve la celda que se ha creado para que pueda ser añadida a la tabla.

```
- (void)tableView:(UITableView *) tableView
didSelectRowAtIndexPath: (NSIndexPath *)indexPath {
```

Finalmente, este método indica a la aplicación qué hacer cuando el usuario pulse una de las filas de la tabla.

```
InfoClubViewController *inf = [[InfoClubViewController alloc] initWithNibName:@"InfoClubViewController" bundle:nil];
inf.inf=[tableList objectAtIndex:indexPath.row];
inf.navbar=[self navigationItem];
```

En los favoritos, queremos que al pulsar una de las celdas, se cambie la pantalla a la información del club que el usuario ha pulsado. Por esa razón, creamos una variable del tipo InfoClubViewController, que es la ventana que queremos abrir, y la inicializamos. Escogemos el initWithNibName como constructor, porque el parámetro que le pasamos es el fichero .xib que debe abrir, que como se ha comentado en la parte practica, es donde reside la información gráfica de la aplicación. Después de crear la variable, le pasamos la información que necesita para que sepa que club se selecciono.

```
 [self.navigationController pushViewController:inf animated:YES];
 [inf release];
```

Para poner otra pantalla, se llama el método pushViewController del navigationController. Haciendo esto, aparece la nueva pantalla por la derecha deslizándose, si se ha pedido que sea animada. Este tipo de presentación sirve para mostrar nuevas pantallas que pertenecen a la anterior. De esta manera se puede mostrar una jerarquía de la información de manera clara.

Hay otra forma para mostrar una nueva pantalla, y es hacer una presentModalViewController, que es hacer que se muestre la nueva vista aparecer desde debajo, y tapando todo. Esta ultima sirve para cuando se quiere mostrar una información relacionada con la pantalla, pero que no es una subcategoría de esta, por ejemplo, serviría para mostrar unos ajustes en una pantalla, que no son una subcategoría de aquella, pero si que pertenecen a la misma.

Una vez hemos implementado los métodos para que funcione, falta especificar el aspecto grafico de la tabla. Abrimos el interface builder a partir del fichero .xib que se creó al crear las clases. Una vez ahí, ponemos una tabla en la vista, y en el connections inspector, enlazamos los atributos dataSource y delegate de la tabla con la clase, igual que con la variable de la clase que nos hemos declarado, del tipo UITableView, la enlazamos con la tabla. De esta forma le indicamos con que tabla tiene que coger los atributos, cual es la clase que maneja los datos.

En este momento ya tenemos la tabla posicionada, y le hemos indicado quien la controla. Pero, aún así, tenemos que añadir mas funcionalidades a dicha implementación. Como es una lista de favoritos, tenemos que poder añadir, quitar clubs, y modificar su posición.

Para la inserción de nuevos clubs en los favoritos, se explicará en el apartado 3.6, referente a la información del club, pues es ahí donde se ha implementado.

```
[mainTableView setEditing:!mainTableView.editing animated:YES];
```

Este método sirve para permitir editar los clubs. Se enlaza una función con un botón, y dentro de la función se llama a este método, que pone la tabla en modo editable, con lo que aparecen uso botones rojos con una barra horizontal blanca, que sirve para borrarlos.

```
-(void) tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle) editingStyle forRowAtIndexPath:
(NSIndexPath *)indexPath{
```

Éste es el método que se llama cuando se pulsa el botón de borrar. En él, tenemos que definir que hacer. En este caso, eliminar la celda de la tabla, del array, y guardarlo en el fichero.

```
[data removeObjectAtIndex:indexPath.row];
```

```
[tableView                          deleteRowsAtIndexPaths:[NSArray
arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationLeft];
```

El método define que índice se ha pulsado, con ello podemos saber cual se borró. Primero borramos el objeto del array, pues se introdujeron por el mismo orden. En segundo lugar, se borra el objeto de la tabla. Como se pide un array, por si se quieren borrar mas elementos, creamos un array con el índice. En este método definimos que haya una animación, y es que se borre yendo a la izquierda.

```
[data writeToFile:[self dataFilePath] atomically:YES];
```

La información se ha borrado de la aplicación, pero sigue en el fichero, es por eso que llamamos a este método para guardar la información. La clase NSMutableArray tiene un método propio para guardar información, que es writeToFile. Con este simple método guardamos la información al fichero que le decimos, que en este caso llamamos al a función dataFilePath, que nos da la ruta al fichero del principio, para que guarde encima. De este modo, cuando se vuelva a abrir la aplicación se mantendrán los cambios, que de otro modo no pasaría.

```
-(BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:
(NSIndexPath *) indexPath {
    return YES;
}
```

Después de poder borrar, tenemos la posibilidad de poder mover los elementos. Con este método, le indicamos a todas las celdas que se pueden mover. Si por alguna razón quisiéramos que alguna celda no se pueda mover, nos pasa el índice para poder comparar y devolver no. De esta forma, esa no se podrá mover.

```
-     (void)     tableView:     (UITableView     *)     tableView
moveRowAtIndexPath:(NSIndexPath                *)sourceIndexPath
toIndexPath:(NSIndexPath *)destinationIndexPath {
```

```
NSString *stringToMove=[[data objectAtIndex:sourceIndexPath.row]
retain];
     [data removeObjectAtIndex:sourceIndexPath.row];
     [data                              insertObject:stringToMove
atIndex:destinationIndexPath.row];
     [stringToMove release];
     [data writeToFile:[self dataFilePath] atomically:YES];
}
```

Aunque hayamos programado la función que indica si se puede mover, el sistema busca el método moveRowAtIndexPath, pues si no está, no permite mover la filas, pues no sabría que hacer con ellas. Con este código cambiamos la posición del objeto seleccionado de la posición sourceIndexPath a destinationIndexPath, y para hacerlo bien, nos creamos una variable para guardar el valor mientras lo borramos y lo guardamos. Finalmente, guardamos el resultado en el fichero.

En este estado nos encontramos con un problema. El problema es cuando hemos añadido un elemento, pues se ha añadido en el array de datos, pero la tabla no se ha percatado del cambio, para hacerlo se utiliza la función reloadData de la clase UITableView.

```
- (void) viewWillAppear:(BOOL)animated{
     data          =          [[NSMutableArray          alloc]
initWithContentsOfFile:[self dataFilePath]];
     [mainTableView reloadData];
}
```

Con saber que se tiene que utilizar la función reloadData, no hay bastante, pues se tendría que tener una referencia de la tabla en todos los sitios donde se pudiera tener acceso a la tabla. Para solucionar se utiliza el método viewWillAppear, que se ejecuta siempre que se llama esta clase. De este modo, cuando se pulsa el botón de la tabBar de favoritos, se llama la función, donde se carga la información del fichero otra vez al array, por si ha habido cambios, y se actualiza la tabla.

Una vez se han programado todas las funcionalidades de la aplicación, falta integrar el diseño. Para ello tenemos el fondo de la pantalla y la tabla para diseñar.

```
vista.backgroundColor = [UIColor colorWithPatternImage:[UIImage
imageNamed:@"fondoFavoritos.png"]];
```

vista es la variable que hace referencia la estructura que mantiene a los otros objetos, y donde tiene que ir la imagen de fondo. La única forma que hay para poner una imagen de fondo es esta que se muestra. Como no hay ningún método para añadir una imagen, se utiliza el backgroundColor, pero tenemos la suerte que la clase UIColor tiene el método colorWithPatternImage, que dado una imagen, devuelve un color con el patrón de la imagen. De esta forma se puede poner una imagen como fondo de la vista, sin tener que poner un objeto UIImageView al fondo del todo con la imagen, que cuesta mas de poner, y da problemas ya que no se ajusta perfectamente.

Otro punto para el diseño es como introducir el fondo de las celdas, pues es un degradado de blanco que desaparece. La mejor opción, y es la utilizada en este punto, ha sido poner el degradado con la imagen de fondo y hacer la tabla transparente.

```
myTableView.backgroundColor = [UIColor clearColor];
```

La variable myTableView hace referencia a la tabla para ponerla transparente. Para hacerlo, hacemos igual que el fondo de la vista, utilizando la propiedad backgroundColor, que le damos un color con la variable de transparencia alpha a 0. Para hacerlo de modo fácil, la clase UIColor tiene el método clearColor, que devuelve un color neutro totalmente transparente. De esta manera, hemos conseguido que la tabla sea transparente, por lo tanto se vea el fondo, pero también se vean los contenidos de la misma.

Tanto las vistas como las barras de navegación tienen diseño, por tanto, los botones que en ellas residen tienen que mantener dicho diseño.

```
UIButton *add = [UIButton buttonWithType:UIButtonTypeCustom];
UIImage *image;

image = [UIImage imageNamed:@"botón ajustes.png"];
[add setImage:image forState:UIControlStateNormal];
```

Primero creamos un botón y una imagen, y le damos la imagen que hemos establecido, así tenemos un botón con la imagen dada.

```
add.bounds=CGRectMake(0, 0,image.size.width, image.size.height);
```

Con el atributo bounds, establecemos las dimensiones del botón, que en este caso serán las mismas que la imagen. De esta forma, podemos establecer la imagen como botón, no tiene necesidad de tener el recuadro típico.

```
[add addTarget:self action:@selector(goSettings:)
  forControlEvents:UIControlEventTouchUpInside];
```

Con el método addTarget, se añade una función a un botón. Primero se especifica quien recibe el mensaje, que normalmente se pone el self, que indica la clase misma. En el action se especifica la función que se llamará, y finalmente, cuando se hará, que en este caso es cuando se pulse el botón.

```
self.navigationItem.leftBarButtonItem = [[UIBarButtonItem alloc]
initWithCustomView:add];
[image release];
[add release];
```

Finalmente, añadimos el botón a la parte izquierda de la navigationBar., y ya para terminar, liberamos las variables utilizadas.

En los botones de la navigationBar no se puede mezclar texto con imagen, se tiene que escoger si poner una imagen o un texto, pero no se pueden poner ambos. Esto comporta un problema pues todos los botones tienen un diseño establecido y mayormente con un texto. El hecho de no poder poner un texto en un botón de la navigationBar supone un problema si se quiere hacer la aplicación con multi-idioma, pues el texto del botón tiene que estar añadido en la imagen, y en vez de tener que seleccionar un texto u otro dependiendo del idioma, se tiene que escoger una imagen u otra, lo que supone mas imágenes y por tanto, mayor peso de la aplicación.

Ilustración 10 Favoritos

3.4 Buscar

En los Destacados se muestran las discotecas mas populares, en los clubs las mas cercanas a ti, y en favoritos las que has añadido, pero quedan aquellas que no se pueden mostrar ahí por si no las tienes cerca y no son de las mas importantes. Para poder buscar una discoteca de este tipo existe un buscador por nombre.

En esta vista, se muestra un buscador típico de iPhone donde buscar las discotecas. Mientras se va escribiendo, se muestra una lista con los clubs que coinciden con el nombre escrito.

```
@interface BuscarViewController : UITableViewController
<UISearchDisplayDelegate, UISearchBarDelegate> {
```

Para poder implementar un buscador, se crea una clase que herede de UISearchDisplayDelegate y UISearchBarDelegate. De esta manera podremos implementar las funciones de buscar de la misma.

```
NSMutableArray *listContent;
NSMutableArray *filteredListContent;
```

Luego, en el mismo archivo .h, declaramos una serie de variables que utilizaremos. La listContent, es un array que contiene todos los clubs, es ahí donde buscaremos, mientras que la filteredListContent, contiene el filtrado de los clubs según los parámetros de búsqueda.

```
NSString *fílenme = @"llista.plist";
NSString *filePath = [[NSBundle mainBundle] pathForResource:
fílenme ofType:nil];
listContent= [[NSMutableArray alloc] initWithContentsOfFile:
filePath] ;


self.filteredListContent = [NSMutableArray arrayWithCapacity:
[self.listContent count]];
```

Cuando se carga la aplicación, lo primero que hacemos en la vista de búsqueda es cargar los clubs disponibles tal y como se explicó en la sección de favoritos, y hacemos una copia para la lista filtrada.

Cómo también existe una lista de elementos, que son los posibles resultados de la búsqueda, se tienen que implementar los métodos propios de la lista, tal y como se explicó en el apartado 3.3 Favoritos.

```
-          (NSInteger)tableView:(UITableView          *)tableView
numberOfRowsInSection:(NSInteger)section
{
     if                        (tableView                        ==
self.searchDisplayController.searchResultsTableView)
     {
          return [self.filteredListContent count];
     }else{
          return [self.listContent count];
     }
}
```

La función numberOfRowsInSection, que indica el número de filas que tendrá la lista cambia un poco respecto a la de favoritos, pues depende de si estamos buscando o no, mostrará la información de uno u otro array. Si no estamos buscando, mostrará la información de la listContent, sino, de la filteredListContent. Por tanto, tenemos que devolver el número de elementos respectivos.

```
-    (UITableViewCell    *)tableView:(UITableView    *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
     static NSString *kCellID = @"cellID";

     UITableViewCell         *cell         =         [tableView
dequeueReusableCellWithIdentifier:kCellID];
     if (cell == nil)
     {
```

```
          cell        =        [[[UITableViewCell        alloc]
initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:kCellID] autorelease];
          cell.accessoryType                                    =
UITableViewCellAccessoryDisclosureIndicator;
     }


     NSDictionary *product;
     if                      (tableView                      ==
self.searchDisplayController.searchResultsTableView)
     {
        product           =           [self.filteredListContent
objectAtIndex:indexPath.row];
    }
     else
     {
        product               =               [self.listContent
objectAtIndex:indexPath.row];
    }


     cell.textLabel.text = [product objectForKey:@"club"];
     return cell;
}
```

Lo mismo nos pasa con la función cellForRowAtIndexPath, todo es lo mismo exceptuando cuando tenemos que rellenar la información, que cogemos la información de una lista u otra.

```
-     (void)     tableView:     (UITableView     *)     tableView
didSelectRowAtIndexPath: (NSIndexPath *)indexPath
{
     NSDictionary *product = nil;
     if                      (tableView                      ==
self.searchDisplayController.searchResultsTableView)
     {
        product           =           [self.filteredListContent
objectAtIndex:indexPath.row];
    }
```

```
      else
      {
         product                =                [self.listContent
objectAtIndex:indexPath.row];
     }


      InfoClubViewController   *inf   =   [[InfoClubViewController
alloc] initWithNibName:@"InfoClubViewController" bundle:nil];
      inf.inf= product;
      [self.navigationController           pushViewController:inf
animated:YES];
      [[tableView                cellForRowAtIndexPath:indexPath]
setSelected:NO animated:YES];
      [inf release];
}
```

Igual que en el didSelectRowAtIndexPath, pues si ha seleccionado una celda u otra, tenemos que escoger la información de la lista que corresponda con la información mostrada.

Ahora que tenemos lista la lista, que muestra los resultados filtrados, tenemos que filtrar los resultados.

```
-(void)filterContentForSearchText:(NSString*)searchText
scope:(NSString*)scope
{
```

Esta función se llama cada vez que se modifica el texto de búsqueda. Es aquí donde se cambiará la información mostrada en la lista.

```
[self.filteredListContent removeAllObjects];
```

Primero quitamos todos los objetos del array para poder añadir aquellos que superen las restricciones.

```
for (NSDictionary *product in listContent){
      NSComparisonResult        result        =        [[product
objectForKey:@"club"]                              compare:searchText
```

```
options:(NSCaseInsensitiveSearch|NSDiacriticInsensitiveSearch)
range:NSMakeRange(0, [searchText length])];
     if (result == NSOrderedSame){
          [self.filteredListContent addObject:product];
     }
}
```

Por cada objeto en la lista inicial, comprueba que el nombre del mismo contenga la palabra puesta en el buscador, sin tener en cuenta las mayúsculas ni la posición de la misma. Si lo es, añade este elemento en la lista de contenido filtrado.

```
-(BOOL)searchDisplayController:(UISearchDisplayController
*)controller          shouldReloadTableForSearchString:(NSString
*)searchString
{
     [self filterContentForSearchText:searchString scope:nil];
     return YES;
}
```

Este método llama la función que hay encima de esta para actualizar la tabla de contenidos. Siempre devuelve YES, pues indica que la tabla se tiene que refrescar.

Una vez hemos programado el código, tenemos que poner los elementos del interface builder en su debido lugar i enlazarlos. En el xib general tenemos un navigation controler que nos permite hacer una navegación con navBar. En este apartado pondremos el buscador. Arrastramos a la estructura una SearchDisplayController. Este Controlador nos permite hacer el buscador ya que lleva consigo una searchBar. Esta barra de búsqueda la ponemos dentro de la vista que habremos definido con una tabla, y finalmente el viewController de la vista, tiene que ser de la clase que hemos definido, tal i como se ha definido en la imagen XX

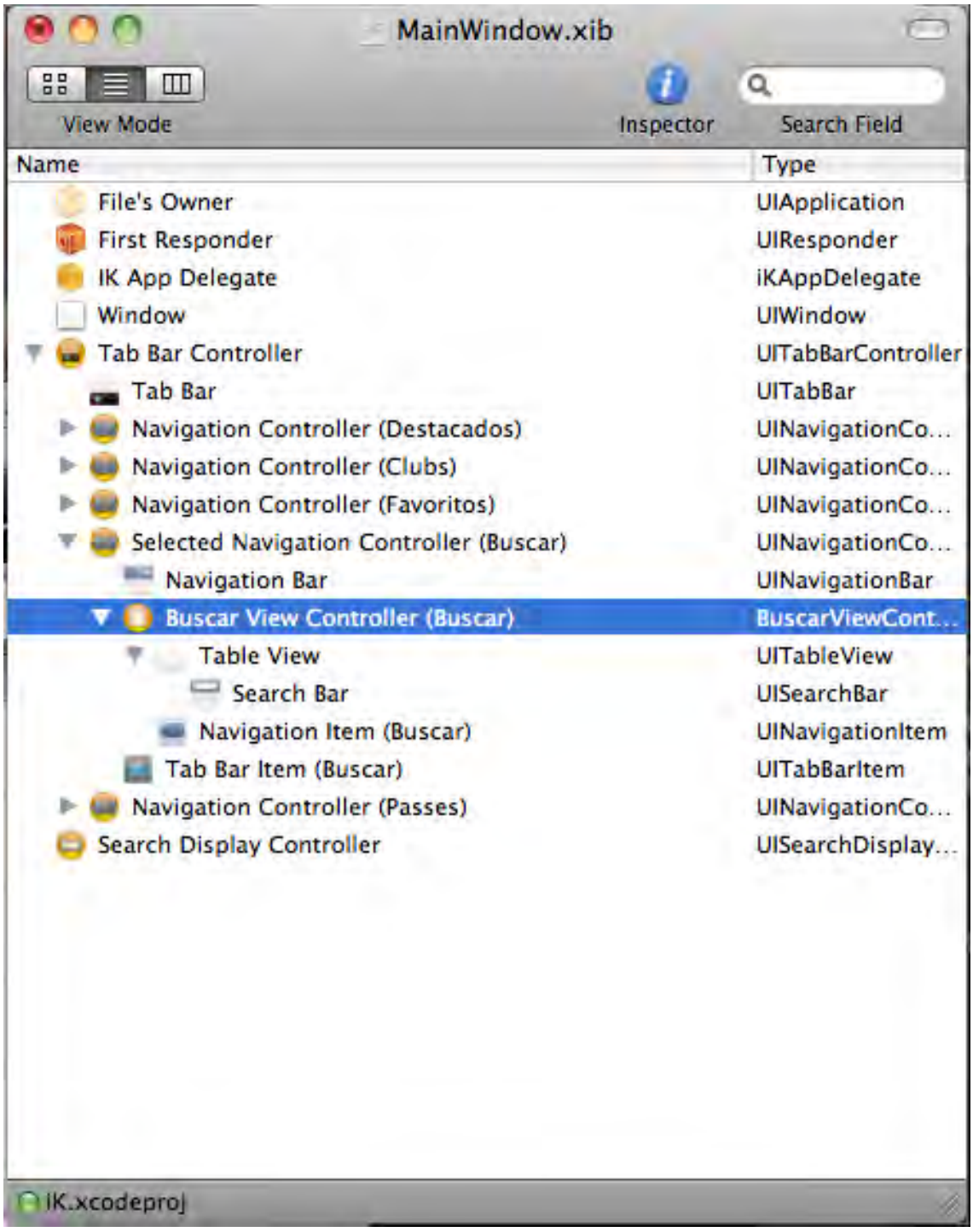

Ilustración 11 Organización buscar

Una vez hemos puesto todos los elementos, tenemos que enlazarlo. Los elementos propios de la tabla, como el dataSource y el delegate, se enlazan con la tabla misma, mientras que los de buscar, se enlazan con el searchDisplayController, al igual que el delegate. Así pues, ya tendremos

todas las funciones y atributos enlazados, al igual que el delegate que tendrá múltiples enlaces.

Con esto ya tenemos hecho un buscador de elementos, pero como en el caso anterior, aún faltan un par de pasos, que son el diseño y la conexión con el servidor.

La integración con el diseño fue fácil, pues este apartado tiene poco diseño. La tabla se utiliza la por defecto, con el fondo blanco, y sin modificación, pues buscar clubs es costoso, y por tanto el servidor solo puede enviar los nombres. Lo único que cambia es el color de la barra de búsqueda.

Para poner una imagen en el buscador, utilizamos el mismo método que en la NavigationBar, reimplementamos el método drawRect indicándole que imagen se tiene que pintar.

Un problema que hay es cuando un club se da de alta en la aplicación o cambia su subscripción, de este modo en el que está hecho, se tendría que poner una actualización para que apareciera el cambio, lo cual seria lento y costoso.

La solución propuesta y utilizada es utilizar la conexión a Internet que ofrece el iPhone para conectarse a un servidor y recoger los datos. Cada vez que el usuario cambia la frase de búsqueda, se envía un mensaje al servidor con dicha palabra o frase, y el servidor devuelve una lista de los clubs que cumplen esa restricción.

Para más información sobre el servidor, consultar la memoria de Valentí Freixanet iKlubbers: Diseño e implementación de una aplicación iPhone - Parte servidor

Ilustración 12 Buscar

3.5 Historial

Tal y como se comentó, la aplicación se basa en el hecho de poder apuntarse a listas VIP para poder entrar gratis en las discotecas, pero para poder demostrar que uno se ha apuntado, existe esta sección, donde se ven los pases que se han adquirido.

Una vez te apuntas a un club, se añade un club al property list del historial, con el nombre del club, el logotipo, el día de utilización y el código. Éste código sirve para poder entrar en la discoteca, pues una vez allí, muestras el iPhone con esta pantalla, y el club, que tendrá una lista, comprobará que ese código está en la lista, y permitirá al usuario de iKlubbers entrar gratuitamente.

También se guardan los cupones de otras veces que se haya entrado en una discoteca de esta forma, pero si el pase ya caducó, es decir, es antiguo, aparecerá una imagen encima diciendo cancelado, para mostrar que ya se utilizó y que está cancelado. Estos pases se ven con un page control, lo cual significa que en la pantalla habrá un solo pase, pero haciendo un gesto con el dedo arrastrando hacia un lado la imagen, se desplazará hacia ese lado, dando paso a otro pase. Estos pases también se pueden mostrar como un listado, para ver todos los clubs que se han visitado.

El page control, que es una forma diferente de ver los pases, que ayuda al usuario porque es muy usable, solo permite ver hasta 21 paginas, que en este caso se traduce a poder ver 21 cupón. Cuando el usuario se ha apuntado a 21 clubs, y quiere apuntarse a otro, se borra el último club y se añade el nuevo, así evitamos tener una lista demasiado larga como historial guardando demasiada información. Para dar mas libertad al usuario, se le permite borrar los cupones pasados y que ya no quiere.

Para programar esto, hemos hecho igual que en las dos últimas, hemos implementado los métodos necesarios para la lista.

```
-    (NSInteger)    tableView:    (UITableView    *)    tableView
numberOfRowsInSection: (NSInteger *) section{
      return [tableList count];
}
```

Primero definimos el número de filas que tendrá la tabla

```
-    (UITableViewCell    *)tableView:(UITableView    *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {


      static NSString *CellIdentifier = @"paseViewCell";


      UITableViewCell   *cell   =   (UITableViewCell   *)[tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
      if(cell == nil){
            cell   =   [[[UITableViewCell   alloc]   initWithStyle:
UITableViewCellStyleDefault      reuseIdentifier:      kCellID]
autorelease];
            cell.accessoryType=
UITableViewCellAccessoryDisclosureIndicator;
      }


      NSDictionary                                *inf=[[tableList
objectAtIndex:indexPath.row]retain];


      cell.text = [inf objectForKey:@"club"];
      [inf release];
     return cell;
}
```

Luego, como todas las tablas, definimos el contenido de cada celda.

```
-     (void)     tableView:     (UITableView     *)     tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
     InfoClubViewController   *inf   =   [[InfoClubViewController
alloc] initWithNibName:@"InfoClubViewController" bundle:nil];
     inf.inf = [tableList objectAtIndex:indexPath.row];
     [self.navigationController           pushViewController:inf
animated:YES];
     [[tableView                cellForRowAtIndexPath:indexPath]
setSelected:NO animated:YES];
     [inf release];

}
```

Finalmente, añadimos la funcionalidad cuando el usuario pulse una celda, que al igual que las otras tablas, cambiará de vista para mostrar la información del club.

Esto es a lo que la tabla se refiere, pero también habrá un page control, que permitirá ver los cupones y desplazarlos con un sencillo gesto del dedo. Cuando se inicializa la aplicación, en el método viewDidLoad, preparamos la información para que se muestre.

```
NSString *fílenme = @"cupon.plist";
NSString  *filePath  =  [[NSBundle  mainBundle]  pathForResource:
fílenme ofType:nil];

tableList=                [[NSMutableArray                alloc]
initWithContentsOfFile:filePath];
[table reloadData];
```

Primero de todo, como siempre, cargamos la información en un array para poder mostrarla, y actualizamos la tabla.

```
kNumberOfPages = [tableList count];
NSMutableArray *controllers = [[NSMutableArray alloc] init];
for (unsigned i = 0; i < kNumberOfPages; i++) {
     [controllers addObject:[NSNull null]];
}
self.viewControllers = controllers;
[controllers release];
```

Luego, en la variable kNumberOfPages, que contendrá el número de paginas del page control, guardamos el número de elementos que hay en la lista. Después creamos otro array para guardar las diferentes pantallas que se mostraran, de las mismas dimensiones que la lista, pues cada elemento estará en una página diferente. Asignamos esta nueva información al array global que tenemos.

```
scrollView.pagingEnabled = YES;
scrollView.contentSize = CGSizeMake(scrollView.frame.size.width
* kNumberOfPages, scrollView.frame.size.height);
scrollView.showsHorizontalScrollIndicator = NO;
scrollView.showsVerticalScrollIndicator = NO;
scrollView.scrollsToTop = NO;
scrollView.delegate = self;

pageControl.numberOfPages = kNumberOfPages;
pageControl.currentPage = 0;
```

Para que el page control funcione bien, se necesitan dos componentes, un page control, que es el que muestra cuantas paginas hay, y en cual estamos, y un scrollView, que es el que muestra la información. Inicializamos los dos para dejarlos preparados para su utilización.

```
[self loadScrollViewWithPage:0];
[self loadScrollViewWithPage:1];
```

Luego cargamos las paginas cuando las necesitamos. En el momento del inicio, necesitamos las dos primeras.

```
- (void)loadScrollViewWithPage:(int)page {
    if (page < 0) return;
    if (page >= kNumberOfPages) return;
```

En la función loadScrollViewWithPage, cargamos la información de una página, y por eso nos dan le número de página se necesita. Si esta página es menor a 0 o superior al número de páginas que existe, sale de la función, pues no existe información disponible para ello.

```
CuponViewController *controller = [viewControllers objectAtIndex:page];
if ((NSNull *)controller == [NSNull null]) {
    controller = [[CuponViewController alloc] initWithPageNumber:page];
    [viewControllers replaceObjectAtIndex:page withObject:controller];
    controller.tname = [[tableList objectAtIndex:page] objectForKey:@"club"];
    controller.tcode = @"P3482DC";
    controller.tdate = @"31/05/10";
    controller.tlogo = [[tableList objectAtIndex:page] objectForKey:@"photo"];
    controller.tcancel = @"selloCancel.png";
    [controller release];
}
```

En la variable viewControllers guardamos la paginas que hemos mostrado, por tanto, cogemos la pagina que necesitamos ahora. Si esta aún no se ha creado, la creamos y le damos la información que necesita para poder visualizarse.

```
if (nil == controller.view.superview) {
    CGRect frame = scrollView.frame;
    frame.origin.x = frame.size.width * page;
    frame.origin.y = 0;
    controller.view.frame = frame;
    [scrollView addSubview:controller.view];
```

```
}
```

Si ese controlador no tiene vista definida, creamos una. Lo que hace es tratar a las vistas del pageController como si fueran una sola vista muy larga, y elige que trozo mostrar.

```
- (void)scrollViewDidScroll:(UIScrollView *)sender {
     if (pageControlUsed) {
          return;
     }
```

Este método se llama siempre que el scrollView se mueve mas del 50% de la vista, y la primera cosa que se tiene que hacer es comprobar si ya se ha activado, con lo que si es cierto, sale del método sin hacer nada mas, pues ya se activó.

```
CGFloat pageWidth = scrollView.frame.size.width;
int page = floor((scrollView.contentOffset.x - pageWidth / 2) / pageWidth) + 1;
pageControl.currentPage = page;

[self loadScrollViewWithPage:page - 1];
[self loadScrollViewWithPage:page];
[self loadScrollViewWithPage:page + 1];
```

Si aún no hemos movido la vista, actualizamos el número de página, y después cargamos las pantallas anterior, actual y después.

```
- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView {
     pageControlUsed = NO;
}

- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
     pageControlUsed = NO;
}
```

Al principio y al final del movimiento del page control, reseteamos la variable pageControlUsed, para indicar que aún no se ha utilizado, y es en la función changPage donde se cambia.

```
- (IBAction)changePage:(id)sender {
     int page = pageControl.currentPage;

     [self loadScrollViewWithPage:page - 1];
     [self loadScrollViewWithPage:page];
     [self loadScrollViewWithPage:page + 1];

     CGRect frame = scrollView.frame;
     frame.origin.x = frame.size.width * page;
     frame.origin.y =0;
     [scrollView scrollRectToVisible:frame animated:YES];

     pageControlUsed = YES;
}
```

Es en esta función donde cargamos la información. Primero cogemos cual es la pagina actual, cargamos las 3 vistas, y finalmente establecemos la vista, en que posición tenemos que mostrarlo. Ponemos la variable pageControlUsed a cierto, para indicar que se esta moviendo.

En este momento tenemos la lista junto con el page control en la misma vista, pero lo que queremos es que solo se muestre uno de los dos, y que al pulsar un botón, se gire la vista mostrando el otro.

```
[UIView beginAnimations:nil context:nil];
[UIView setAnimationDuration:1.0];
[UIView
setAnimationTransition:UIViewAnimationTransitionFlipFromRight
                              forView:[self view]cache:YES];

[[self view] exchangeSubviewAtIndex:1 withSubviewAtIndex:0];
[UIView commitAnimations];
```

Con este código, cambiamos una subview por otra, y por ello, tenemos que tener dos vistas dentro de la misma, tal y como se mostrará mas adelante. De esta forma cambia la segunda vista con la primera, y al hacer el cambio, el índice se cambia, así que no se tiene que cambiar el código para cuando se ha cambiado de vista. Esta función está enlaza con un botón, que también contendrá el código para otros cambios, como cambiar el nombre del mismo botón o el título que aparezca en la

navigationBar.

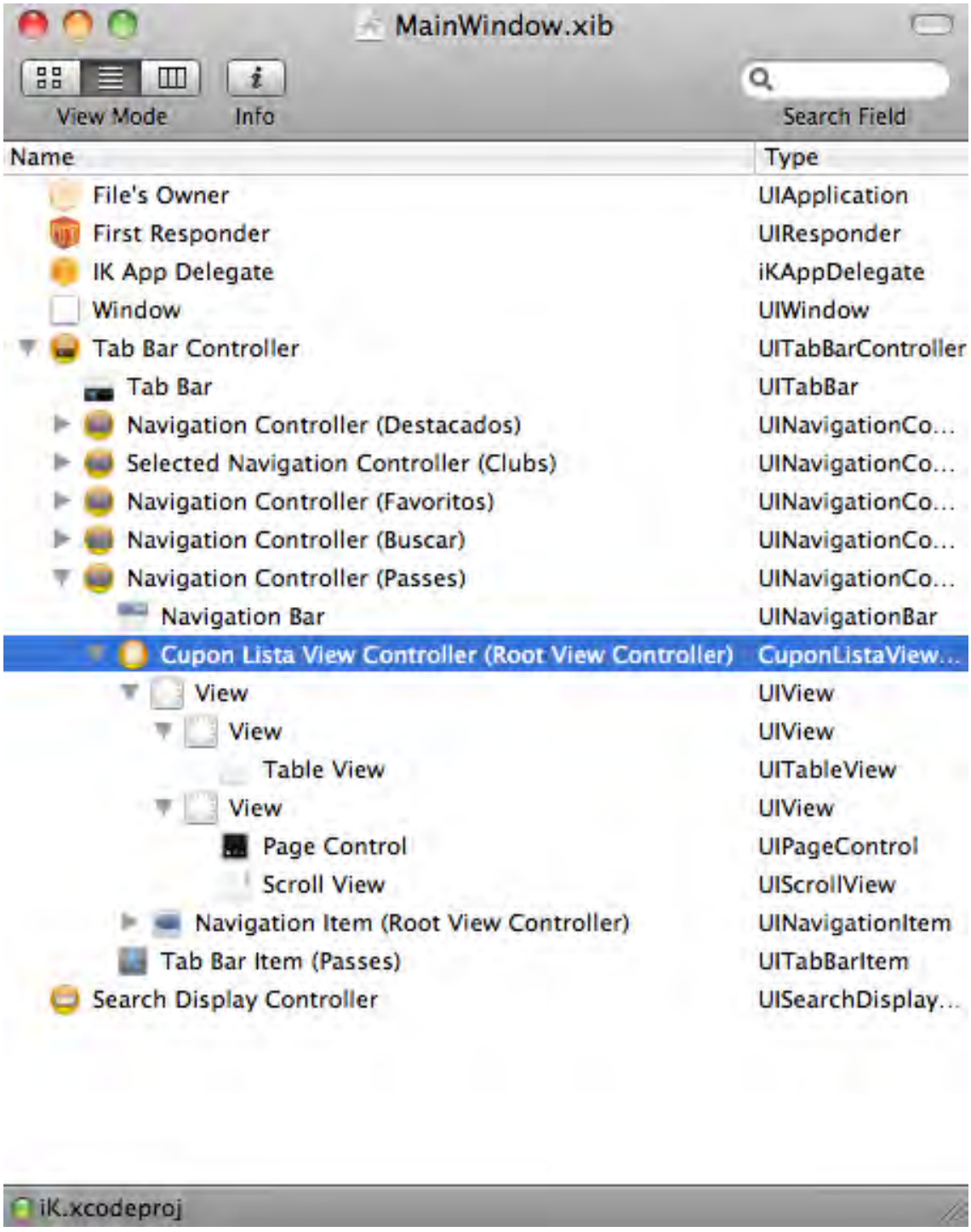

Ilustración 13 Organización cupón

Una vez hecho esto, nos falta enlazarlo con el interface builder tal y como se hizo con las otras, pero teniendo en cuenta, la estructura que se muestra

en la figura, para poder hacer el cambio de vista cuando se pulsa el botón.

Para hacer la integración de diseño, se hizo lo mismo que con los favoritos a lo que la lista se refiere. Para la integración del diseño para la parte de los cupones, se ha hecho de la siguiente manera. Como la pagina del page control es otro archivo que tiene su propio archivo xib, se crea ahí todo el diseño de cada vista. Establecemos el fondo de la vista a invisible y enlazamos el resto de los objetos para que puedan ser dinámicos y que la información sea proporcionada por la clase anterior.

Ilustración 14 Pases (Pases y lista)

3.6 Información del club

Uno de los puntos mas importantes que se tienen que tratar es la información de los clubes, pues es donde el usuario consultará como es el club y donde se apuntará a la lista.

Esta pantalla se ha dividido en los 3 puntos que se describirán a continuación. Aún así, habrá una información común que se mostrará en todas las vistas. Esta información es la básica de cualquier club, que son el nombre, el logotipo, la valoración, y los botones de apuntarse a la lista y el de ver el póster de la noche, así como la información de apertura y cierre. Esta información se guarda en todas las pantallas pues es la mas importante que se debe recordar, y no tener que memorizar donde está para acceder a ella.

La programación de esta pantalla no tiene mucho secreto, se programa igual que las otras funcionalidades descritas anteriormente. Existe una vista con diferentes imágenes, botones y textos, que se asignaran como es de costumbre, y en medio un page control para poder mostrar las diferentes vistas. Este page control varia un poco, pues no se muestra la misma información en todas las paginas, pues en cada una va una información u otra.

En el método donde se carga la nueva página, la función loadScrollViewWithPage, en vez de seleccionar una clase por defecto, se selecciona una utilizando la pagina como identificador, pues tenemos un archivo creado para cada vista.

En la primera se muestra información del club, donde aparece la descripción del club cedida por el mismo club. Si la información supera las dimensiones de la pantalla, aparece una barra lateral que indica que se puede hacer scroll para ver la continuación.

En la tercera página se muestran imágenes en pequeño del local, también proporcionadas por el club. Al seleccionar una de ellas, sube una nueva vista mostrando la imagen en grande.

```
ImageViewController *img = [[ImageViewController alloc]
initWithNibName:@" ImageViewController " bundle:nil];
[self presentModalViewController:img animated:YES];
```

Tal y como se comentó en el apartado de 3.3. Favoritos, para hacer que suba una nueva vista con un contenido se hace con una modalView, y se ejecuta tal y como se muestra en el código superior. Se crea una instancia de la clase que se quiere mostrar, indicando como se llama el archivo xib que corresponde a él. Finalmente, se llama la función propia de la clase llamada presentModalViewController pasándole la clase creada.

Las imágenes serán botones con la imagen en si como imagen, y al pulsar encima de ellas, llamaran una función que contendrá el código que se ha escrito.

Finalmente, la última página del page control, muestra los comentarios que ha hecho la gente de ese club, y donde se puede añadir otro junto con una valoración del club. Para más detalles de esta parte consultar la memoria de Valentí Freixanet, iKlubbers: Diseño e implementación de una aplicación iPhone - Parte servidor dedicada al servidor Web.

Falta añadir, que como se ha comentado, los clientes de esta aplicación son los clubs, y por tanto, son estos quien pagan a la empresa para poder poner su club en la aplicación. Esto conlleva a que hay 4 tipos diferentes de tarifas para los clubs, y cada una permite más información. La forma de plasmar esto, es que en esta sección de información se mostrará mas contenido o menos según su tarifa, ya que desde servidor se enviará un listado completo de qué información mostrar, y puede que algunas de estas paginas no se muestre en su totalidad, o no estén en algunos clubs.

En todas las ventanas se podrá ver un botón llamado apuntarse, que muestra otra ventana utilizando una modal view, donde el usuario puede apuntarse a una lista indicando el número de acompañantes que habrá. Este desarrollo se ha hecho utilizando las herramientas antes explicadas.

Para terminar esta sección, se ha añadido un botón en la navigationBar que sirve para apuntar ese mismo club a favoritos. De esta forma el usuario podrá incluir el club a su lista de favoritos y poder acceder a él sin necesidad de estar cerca o buscarlo en el buscador.

```
- (IBAction)addAction:(id)sender
{
     NSMutableArray *array;

     NSString *fílenme = @"favourites.plist";
     NSString *filePath=[[NSBundle mainBundle] pathForResource:
fílenme ofType:nil];

     array = [[NSMutableArray alloc] initWithContentsOfFile:
filePath];
     [array addObject:[self inf]];
     [array writeToFile:filePath     atomically:YES];
     [array release];
}
```

Esta función es la que añade el club a favoritos. Para hacerlo, añadiremos el club en la lista, y así, cuando vayamos a la vista de favoritos, volverá a cargar la información con este nuevo. Así pues, primero abrimos el fichero y lo cargamos en una variable, luego añadimos un objeto a dicho array. Este objeto es el que contiene toda la información del club, pero para añadir solo la parte que nos interese, creamos una variable del tipo NSDictionary, y le añadimos los campos que creamos oportunos. Finalmente, utilizando las funciones propias del NSMutableArray, escribimos los datos en el fichero, con el nuevo parámetro añadido. Cuando un club se apunta a una lista, se sigue el mismo procedimiento que el ahora explicado. Se abre el fichero

que contiene los clubs a los que se ha ido, y se añade otro objeto con la información requerida.

```
[array insertObject:[self inf] atIndex:0];
```

Esta es la pequeña variación que hay con añadir a favoritos, y es que esta tiene que ponerse primera, pues es mas importante el pase de hoy que el de un día anterior. Con esta función indicamos a que índice se guarda la información, y como la queremos al principio, el número será 0, pues los objetos empiezan a contarse a partir del 0.

A parte de añadir un club en la lista de pases, se pueden borrar. En este caso se tiene que diferenciar entre las dos vistas, pues cuando se está en vista de tabla, puedes seleccionar el que quieres borrar, mientras que en vista de cupón se borra el presente.

```
UIButton *add = [UIButton buttonWithType:UIButtonTypeCustom];
UIImage *image;

image = [UIImage imageNamed:@"botonBasura.png"];
[add setImage:image forState:UIControlStateNormal];

add.bounds=CGRectMake(0,0,image.size.width, image.size.height);
[add addTarget:self action:@selector(deletePass:)
  forControlEvents:UIControlEventTouchUpInside];

self.navigationItem.leftBarButtonItem = [[UIBarButtonItem alloc]
                                  initWithCustomView:add];
[image release];
[add release];
```

De esta forma cargamos un botón en la navigationBar con la imagen de una basura para indicar que se puede borrar dicho cupón.

```
CGFloat pageWidth = scrollView.frame.size.width;
```

```
int page = floor((scrollView.contentOffset.x - pageWidth / 2) / pageWidth) + 1;
[tableList removeObjectAtIndex:page];
```

Para poder borrar una pagina del cupón, es necesario saber en que página estamos. Por tanto, primero de todo, obtenemos el número de página. Luego quitamos el elemento de la lista.

```
NSString *fílenme = @"cupon.plist";
NSString *filePath = [[NSBundle mainBundle] pathForResource: fílenme ofType:nil];
[tableList writeToFile:filePath atomically:YES];
```

Después de borrar el club de la lista, tenemos que actualizarlo. Buscamos la ruta del fichero, y gravamos ahí el nuevo cambio. Una vez se ha guardado en el fichero, procedemos a actualizar la vista. Una vez hecho esto, se ha actualizado el fichero donde residen los datos, pero la aplicación no ha actualizado las pantallas. Para hacerlo, se llama el mismo código que se puso en el método viewDidLoad, que vuelve a cargar el fichero y las pantallas.

Un problema importante que apareció en ese momento, es que al añadirse a una lista, este añade dicho club a la lista de pases, y al volver a la vista de cupón, al tener el fondo transparente, pone el nuevo cupón encima del anterior, al ser este diferente. Esto crea un problema pues se ven los nombres superpuestos.

```
for (UIView *view in scrollView.subviews) {
      [view removeFromSuperview];
}
```

Para solucionarlo, se eliminan todas las subsistas de un objeto. Con esto conseguimos dejar limpia de vistas al objeto, y con ello no tener el problema que antes se comentaba, pues se van acumulando las vistas.

Ilustración 15 Información (descripción y fotos)

Ilustración 16 Información (comentarios)

3.7 CustomCell

Ya para finalizar, en cada tabla que hay en la aplicación necesita de un diseño propio, y con unos datos diferentes en cada caso. Por defecto, las celdas de una tabla permiten poner un nombre y una imagen en cada una de ellas, pero si se quiere hacer otro diseño, se tiene que crear una celda propia.

```
@interface ClubViewCell : UITableViewCell {
      UILabel *name;
      UILabel *dist;
      UIImageView *val;
      UIImageView *free;
      UIImageView *accessory;
}
@property(nonatomic,retain) IBOutlet UILabel *name;
@property(nonatomic,retain) IBOutlet UILabel *dist;
@property(nonatomic,retain) IBOutlet UIImageView *val;
@property(nonatomic,retain) IBOutlet UIImageView *free;
@property(nonatomic,retain) IBOutlet UIImageView *accessory;
```

Primero de todo, creamos una clase nueva, y en vez de heredar de UIViewController, se hereda de UITableViewCell. Dentro de la misma, se ponen los objetos que se van a mostrar en las celdas. El "property" implementa los métodos get y set de cada atributo, sin necesidad de implementarlos. La propiedad IBOutlet, nos permite enlazar el atributo con su respectivo objeto en el interface builder.

Una vez en el interface builder, quitamos la vista que viene por defecto y añadimos una celda para editarla. En esta vista, ponemos las imágenes y textos que se necesitan.

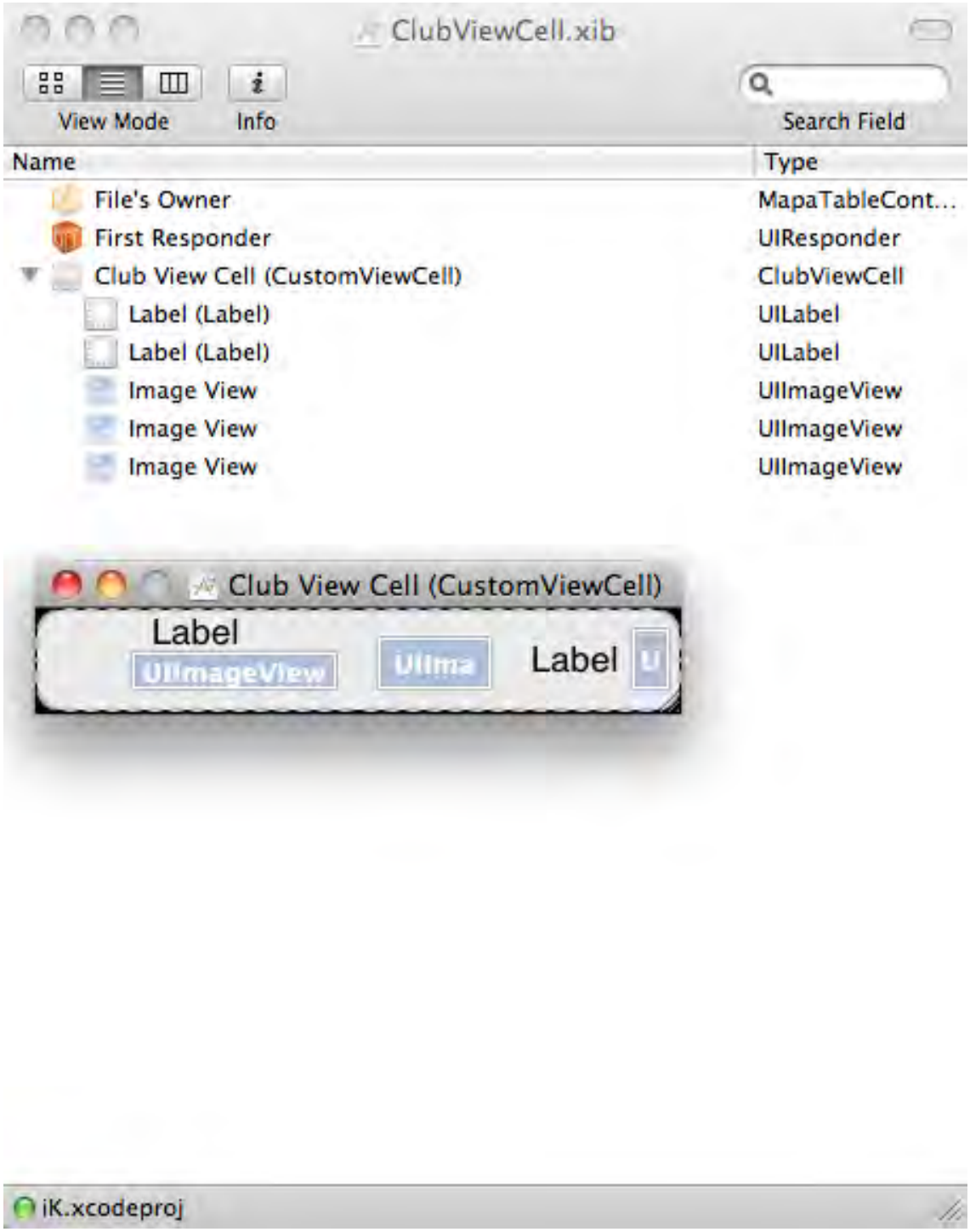

Ilustración 17 Organización de un customCell

Antes de continuar, se cambia la clase de la celda a UITableViewCell, y el File's owner, el propietario del archivo, como la clase que contiene dicha celda, en este caso MapaTableViewCell. Luego, se enlazan los objetos que

hay en la celda, con los atributos implementados que se encuentran en dicha celda.

Para poder implementar una celda en una tabla, se hace lo siguiente. Primero se crea un atributo del tipo ClubViewCell, la clase que hemos implementado, y volvemos al interface builder y enlazamos el atributo ahora creado con la celda para indicar qué celda pertenece a la tabla.

```
ClubViewCell *cell = (ClubViewCell *)[tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
if(cell == nil){
     [[NSBundle mainBundle] loadNibNamed:@"ClubViewCell"
owner:self options:nil];
     cell = tmpCell;
     self.tmpCell = nil;
}
```

Para poder implementar esta nueva celda, en la función cellForRowAtIndexPath, cambiamos la celda al tipo de la clase que hemos creado, ClubViewCell en nuestro caso. A continuación utilizamos la clase NSBundle para cargar el archivo xib. Este archivo se cargará en el atributo creado para enlazar en el interface builder. El siguiente paso, lógicamente, es cargar este nuevo contenido a la celda creada en la función, y establecemos el contenido del atributo interno a null, para dejarla preparada para la siguiente celda.

Un momento la hemos creado, solo queda llenarla con el contenido necesario, ya sea con los textos o imágenes. Un apunte interesante, es que se ha añadido un accessory propio. Para hacerlo, se puso una imagen en la ubicación del accessory, y si es necesario, cargamos la imagen propia, quitando el accesory que puede poner el propio programa.

4 Conclusiones

Durante el desarrollo de esta aplicación se ha aprendido a utilizar un lenguaje de programación nuevo, el Objective C, a utilizar las herramientas de programación XCode como a usar una metodología ágil de trabajo como el SCRUM[47].

Por lo que al Objective C respeta, es una forma de programar un poco diferente a los otros sistemas pues tiene una sintaxis un tanto diferente a los otros lenguajes en algunos aspectos, pero con el tiempo uno consigue entender su funcionamiento y termina por aprender a utilizarlo.

El XCode, juntamente con el interface Builder, es una herramienta realmente diferente a las otras. Cuando se diseña la aplicación con el interface Builder, no aparece el código de aquello, como sucede con otros editores. El hecho de no ver el código de aquello que se ha diseñado, confunde, pues un programador entiende de líneas de código, y es mas fácil que entienda un error cuando este aparece. Ya no solo esto, sino que además se tiene que entender como interactúan el uno con el otro. Todo eso hace que sea mas difícil aprender a hacer el funcionamiento de las herramientas, pero una vez entendido, y se ha aprendido esta nueva filosofía, se ve la facilidad de uso.

El mayor problema que apareció, fue al principio, al juntar el código de mapa con la tabla. La principal característica de juntarlo, fue que las dos tenían que estar en la misma pantalla, pero mostrando una de las dos, y cambiar de una a otra con un movimiento circular. Lograr este efecto, no solo era cuestión de código, pues se encuentra fácilmente en Internet, la dificultad estaba en el interface builder, que se tenían que poner las dos funcionalidades bajo una misma vista, y mostraba una u otra. Fue mas difícil de encontrar la solución, porque no es una forma habitual de trabajar para un programador. La curva de aprendizaje es elevada.

La metodología SCRUM, explicada en la memoria de Marta Cortiñas iKlubbers: Diseño e implementación de una aplicación iPhone – Producción, es una metodología útil y funcional. Ayuda a centrarse en un trabajo, sin tener que pensar en el resto del proyecto. También aporta una visión global del proyecto, pues las reuniones se hacen con todos los integrantes, y se evalúan todos los aspectos de los trabajos a hacer. Éstos métodos dan mas cohesión entre los miembros del grupo, pues todos discuten de los trabajos de todos, así no es un trabajo individual.

Es difícil llegar a complacer a todos los participantes en el proyecto. En este tipo de proyecto se ha tenido dos clientes. Mobivery por una parte pedía unos resultados y unos requisitos en cuanto a la aplicación, mientras que los ponentes de la universidad requerían unos productos promocionales tales como póster, half-sheet, video promocional entre otros. Al plantear el grupo como una empresa, donde cada uno tenga un papel, y uno fuera de productor, pues conseguía que cada uno se centrara en su cometido, y este solucionaba los problemas con los dos clientes, y organizaba las tareas, dando a los programadores la tranquilidad para el desarrollo.

Al ser un grupo multidisciplinar, se tienen que combinar diferentes perfiles, esto hace que personas con diferentes formas de trabajo se tengan que juntar para trabajar y entenderse, pues un programados siempre ha trabajado con programadores pero no con diseñadores, y es diferente comunicar un concepto a alguien que piensa mas como uno, en vez de alguien totalmente distinto.

5 Líneas de futuro

La parte realizada es solo la parte alfa del proyecto. En la parte de planteamiento del proyecto se especificaron muchas funciones que no se pueden añadir.

La primera funcionalidad que se tendría que aplicar seria una seguridad en el servidor. Cuando un club quiere añadirse a iKlubbers o quiere cambiar cualquier cosa de su información, es te tiene que enviar un correo electrónico con los cambios a la empresa y ésta hacer los cambios. Si se pone seguridad en el servidor, el club propio podría cambiar su información de manera automática usando una palabra de paso. Esto daría mas dinamismo a la aplicación pues los clubs podrían cambiar de manera automática la información que quisieran.

Otro paso de futuro cuasi inmediato es colgar la aplicación al appStore. El appStore es donde residen todas las aplicaciones del iPhone, y estas están disponibles para quien quiera descargársela. Según la planificación de tiempo no habrá tiempo de colgar la aplicación pues se tienen que conseguir un número de clientes mínimo para poder colgar la aplicación, sin tener en cuenta que Apple revisa todas las aplicaciones para que sigan sus instrucciones y tardan un promedio de dos semanas en hacerlo.

Unas funcionalidades internas del iPhone que se pensaron que podrían estar, pero que por coste de desarrollo no se pueden implementar dentro del tiempo establecido, son conexión con el faceBook, herramienta social muy extendida hoy en día, que publicaría en qué listas te has apuntado. Junto con esta estaría la posibilidad de ver en tiempo real dentro del local. Mediante una camera Web, se podría ver el interior del local en tiempo real desde cualquier sitio. Otra funcionalidad pensada seria poder escuchar parte del repertorio musical del local para poder ver que tipo de música se escucha y si es del gusto del usuario.

Durante el desarrollo del master se ha llevado a cabo un plan de marketing par poder promocionar la aplicación. Éste se ha diseñado pero no implementado, y es por eso que una buena línea de futuro sería llevarlo a cabo.

Ya finalmente, queda mencionar que entre la entrega de esta memoria y la presentación final se va a desarrollar una funcionalidad mas que va a ser de realidad aumentada. Ésta funcionalidad servirá para poder dejar mensajes en un sitio, y mediante la cámara del iPhone y la aplicación se podrán ver estos mensajes, incluso otros dejados por otros. En la presentación final se va a entregar un anexo con la información detallada.

6 Referencias

[1] iPhone - http://www.apple.com/es/iphone/

[2] iKlubbers - http://www.iklubbers.es/

[3] Mobivery - http://www.mobivery.com/

[4] Master en Creación, Diseño y Engineria Multimédia - http://www.salle.url.edu/portal/masters/masters-multimedia-mcdem-barcelona-presentation

[5] Giroscopio - http://es.wikipedia.org/wiki/Giróscopo

[6] Apple - http://www.apple.com/

[7] XCode - http://developer.apple.com/technologies/tools/xcode.html

[8] lenguaje de programación - http://es.wikipedia.org/wiki/Lenguaje_de_programación

[9] Objective C - http://en.wikipedia.org/wiki/Objective-C

[10] sistema operativo - http://es.wikipedia.org/wiki/Sistema_operativo_móvil

[11] Android - http://es.wikipedia.org/wiki/Android

[12] bliquo - http://www.bliquo.es/

[13] ArounMe - http://www.tweakersoft.com/mobile/aroundme.html

[14] teléfono inteligente - http://es.wikipedia.org/wiki/Telefono_inteligente

[15] Internet - http://es.wikipedia.org/wiki/Internet

[16] pantalla tactil - http://es.wikipedia.org/wiki/Pantalla_tactil

[17] multitáctil - http://es.wikipedia.org/wiki/Multitactil

[18] hardware - http://es.wikipedia.org/wiki/Hardware

[19] minimalista - http://es.wikipedia.org/wiki/Minimalista

[20] megapíxel - http://es.wikipedia.org/wiki/Pixel

[21] software - http://es.wikipedia.org/wiki/Software

[22] widgets - http://es.wikipedia.org/wiki/Widget

[23] sensores - http://es.wikipedia.org/wiki/Sensor

[24] acelerómetro - http://es.wikipedia.org/wiki/Acelerometro

[25] Mac OS X - http://es.wikipedia.org/wiki/Mac_OS_X

[26] powerVR - http://es.wikipedia.org/wiki/PowerVR

[27] iTunes Store - http://es.wikipedia.org/wiki/ITunes_Store

[28] App Store - http://es.wikipedia.org/wiki/App_Store

[29] Safari - http://es.wikipedia.org/wiki/Safari_(navegador)

[30] AJAX - http://es.wikipedia.org/wiki/AJAX

[31] smallTalk - http://es.wikipedia.org/wiki/Smalltalk

[32] Brad Cox - http://en.wikipedia.org/wiki/Brad_Cox

[33] StepStone - http://www.stepstone.com/

[34] NEXTSTEP - http://en.wikipedia.org/wiki/NeXTSTEP

[35] GCC - http://gcc.gnu.org/

[36] POO - http://es.wikipedia.org/wiki/Programación_orientada_a_objetos

[37] herencia múltiple - http://es.wikipedia.org/wiki/Herencia_multiple

[38] API - http://es.wikipedia.org/wiki/Interfaz_de_programación_de_aplicaciones

[39] IDE - http://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado

[40] interface Builder - http://developer.apple.com/technologies/tools/xcode.html

[41] compilar - http://es.wikipedia.org/wiki/Compilador

[42] SDK - http://es.wikipedia.org/wiki/Kit_de_desarrollo_de_software

[43] depurar - http://es.wikipedia.org/wiki/Depurar

[44] breakpoint - http://en.wikipedia.org/wiki/Breakpoint

[45] iPhone Simulator - http://developer.apple.com/technologies/tools/xcode.html

[46] NavBar y tabBar - http://developer.apple.com/iphone/library/documentation/userexperience/conceptual/mobilehig/SpecialViews/SpecialViews.html

[47] SCRUM - http://es.wikipedia.org/wiki/Scrum