UNIVERSIDAD POLITECNICA DE VALENCIA

Tesis del Master de Ingeniería de Computadores

Desarrollo de una aplicación distribuida para dispositivos iOS

Alumno: Javier Querol Morata
Director: Enrique Hernández Orallo

Departamento de Informática de Sistemas y Computadores

Fecha: Diciembre de 2011

1. Introducción

El objetivo de este proyecto ha sido la creación de una aplicación móvil que sea capaz de proporcionar toda la información necesaria en torno a las fallas, todo ello con un enfoque plenamente turístico.

El propósito de la aplicación es unificar toda la información que se encuentra dispersa y presentarla de una forma útil, sencilla y rápida en la palma de la mano de forma instantánea y localizada. De este modo, un usuario con total desconocimiento es capaz de recibir toda la información que necesita, incluso sin saber exactamente que es lo que está buscando.

Haciendo un ligero estudio de mercado, observamos la necesidad de una aplicación de estas características, ya que ninguna de las existentes aprovecha las tecnologías que ofrecen los dispositivos móviles de hoy en día, como puede ser la localización, el geoposicionamiento, o la realidad aumentada.

Siendo las Fallas el principal reclamo turístico de la ciudad de Valencia, parece evidente que haya que aprovechar la tecnología para ofrecer un servicio a tal cantidad de potenciales usuarios finales. El hecho de que este evento sea tan focalizado tanto a nivel temporal, como a nivel geográfico, hace que los dispositivos móviles sean la plataforma perfecta para la implementación de dicha aplicación.

Se ha escogido el dispositivo iPhone de Apple para el desarrollo de la aplicación. La razones por la que se ha elegido este dispositivo son varias, primero porque es uno de los mejores terminales que existe a día de hoy en el mercado, gracias a todas las funcionalidades que ofrece, además la aplicación también es compatible con el dispositivo iPod Touch ya que usa el mismo sistema operativo. También hay que tener en cuenta la sencillez a la hora de difundir las aplicaciones a través de la AppleStore y sobretodo la gran cantidad de usuarios de que existe a su alrededor. Y aunque los dispositivos basados en el sistema operativo Android copan cada día mayor cuota de mercado, los niveles de usabilidad y el porcentaje de usuarios que utilizan las aplicaciones de forma regular son menores. Por lo que a día de hoy aun consideramos iOS la plataforma idónea para la distribución de nuestra aplicación.

Una vez establecido iOS como plataforma de desarrollo, vamos a analizar las distintas tecnologías de las que vamos a hacer uso, como la consulta de información a servidores externos, el uso de realidad aumentada, servicios de localización, notificaciones Push o estadísticas por dispositivo.

En cuanto a la implementación, se ha decido estructurar la aplicación mediante una barra de pestañas. Esta barra de pestañas dispondrá de cuatro vistas principales, cada una con una funcionalidad distinta, además existen subvistas dentro de las vista principales, dotando a la aplicación del contenido necesario. A lo largo del documento se especifican los detalles de implementación acompañados por diagramas que muestran en detalle la estructura de la aplicación.

También se analizan las herramientas de depuración existentes en torno a Xcode así como las pruebas realizadas para comprobar el correcto funcionamiento de la aplicación.

Motivación

Las razones que me han llevado a realizar este proyecto han sido varias, primeramente un interés personal en el dispositivo iPhone de Apple y en la realización de aplicaciones para éste, ya que opino que es un dispositivo interesante y novedoso que ofrece muchas posibilidades. También el hecho de aprender y empezar a adentrarme en una nueva tecnología de desarrollo totalmente desconocida para mi, tanto la plataforma, como el lenguaje de programación y el SDK.

Objetivos

Los objetivos básicos de este proyecto son dos: primeramente aprender una lenguaje y tecnología de desarrollo totalmente nuevos y segundo desarrollar una aplicación para este nueva plataforma que ofrezca a los usuarios información actualizada y de manera cómoda sobre aspectos relacionados con las fallas: como un listado de las fallas ordenado por proximidad, información específica de cada falla, localización en un mapa de las diversas fallas, impresión de las fallas sobre realidad aumentada e información sobre los eventos y servicios próximos al usuario. Es decir, una aplicación que aporte una mejora a la experiencia del usuario.

Alcance

En este proyecto se pretende implementar la aplicación Fallas para los dispositivos iPhone, iPod Touch y iPad. Una aplicación que de soporte a los usuarios turistas o no turistas ofreciendo, de manera siempre actualizada, un conjunto de información que le sea de gran utilidad y que pueda consultar en cualquier momento y lugar, una aplicación que haga mucho más agradable su estancia en Valencia durante la semana fallera.

La aplicación Fallas es en sí un conjunto de siete aplicaciones: Fallas (listado), Fichas, Noticias, Mapa, Realidad Aumentada, Eventos y Servicios.

La aplicación **Fallas** (listado) ofrece una lista con todas las fallas de categoría especial y de Primera A ordenadas de menor a mayor en función de la distancia entre la falla y el dispositivo. Cada celda del listado presenta el nombre de la falla, el nombre de la fallera mayor, la categoría a la que pertenece, el escudo de la comisión y la distancia entre el dispositivo y la propia falla. Aunque la vista por defecto muestra el listado ordenado por distancia, se ofrece la posibilidad de mostrar el listado en función de su categoría.

La aplicación **Ficha** ofrece una vista con la información correspondiente de la falla seleccionada, esta selección se produce a través de la aplicación Fallas (listado) o seleccionando el pin correspondiente en la aplicación Mapa. La información que incluye es la que consideramos necesaria para la satisfacción del usuario, dicha información estaría compuesta por: escudo de la comisión, nombre de la falla, calle de la falla, año de creación, premio 2012 de la falla, premio 2011 de la falla, presupuesto de la falla, texto descriptivo del monumento, imagen del boceto de la falla, foto de la fallera mayor, nombre de la fallera mayor, nombre del presidente, nombre del artista fallero, premio 2012 de la falla infantil, premio 2011 de la falla infantil, presupuesto 2011 de la falla infantil, texto descriptivo de la

falla infantil, imagen del boceto de la falla infantil, foto de la fallera mayor infantil, nombre de la fallera mayor infantil, nombre del presidente infantil y nombre del artista infantil.

La aplicación **Noticias** ofrece una vista con la última noticia más reciente. El dispositivo recibirá notificaciones Push cuando la sección de noticias se actualice.

La aplicación **Mapa** ofrece una vista de mapa y sobre él, una serie de pines que representan las fallas, diferenciados por color en función de su categoría. Si se presiona sobre un Pin se muestra un globo emergente con el nombre de la falla, su escudo, su categoría y la posibilidad de acceder a su ficha correspondiente.

La aplicación **Realidad Aumentada** muestra en pantalla las imágenes capturadas por la cámara trasera y el nombre de la falla sobreexpuesto sobre dichas imágenes. La posición y tamaño del nombre de la falla irá cambiando en función del movimiento registrado por los sensores del dispositivo.

La aplicación **Eventos** ofrece un listado ordenado por días con los eventos más relevantes de las fallas, en cada ficha del evento se proporciona mayor información como la hora del evento, el lugar y una descripción precisa.

La aplicación **Servicios** ofrece un listado categorizado por tipos, donde se muestran los servicios más relevantes y la distancia hasta ellos. En la ficha de cada servicio se aporta información del tipo: nombre, teléfono, dirección y descripción.

2. Estado del arte

En la actualidad existen diversas tecnologías y plataformas que cumplen con los requisitos requeridos para llevar a cabo aplicaciones como la que nos concierne.

Si analizamos con detenimiento los requisitos de la aplicación, podremos determinar con mayor fiabilidad que tecnologías y plataformas se adaptan mejor a nuestros requerimientos. Dado que la aplicación trabaja con información de consulta muy localizada a nivel geográfico, podemos determinar que el dispositivo indicado para la implementación de nuestra aplicación requiere de un sistema de localización relativamente fiable. Aunque los servicios web existentes en la actualidad son capaces de determinar nuestra posición basándose en diferentes aspectos, creemos que los ordenadores no son los dispositivos indicados para esta aplicación ya que se requiere que la información sea ofrecida con unos niveles de movilidad que los ordenadores no son capaces de ofrecer. Por lo que llegamos a la conclusión de que el dispositivo indicado para la implementación de nuestra aplicación es un dispositivo móvil.

Plataformas móviles

Si analizamos con mayor profundidad los diferentes sistemas operativos para dispositivos móviles, llegamos a la conclusión de que existen cuatro candidatos reales a albergar nuestra aplicación.

1. Windows Phone

El más reciente de los sistemas operativos para dispositivos móviles. El hecho de haber sido el último en llegar ha hecho que Microsoft haya aprendido de los errores cometidos por otros sistemas o incluso por ellos mismos en el pasado.

En sus inicios fue planteado como una evolución de Windows Mobile, aunque finalmente el proyecto fue cancelado en 2008. Fue entonces cuando Microsoft realizó una profunda renovación en el grupo encargado del desarrollo de Windows Mobile, cuyo propósito sería la creación de un nuevo sistema operativo. El proyecto finalmente salió a la luz en 2009, y como consecuencia de su rápido desarrollo no se pudo implementar la compatibilidad con Windows Mobile.

Como resultado tenemos un sistema operativo robusto, que comparte gran parte de sus entrañas con el todavía en desarrollo Windows 8. Como características principales, podemos decir que se trata de un sistema cerrado como viene siendo habitual en la compañía de Redmond.

Han basado el sistema en la sencillez y en una interfaz adecuada para dispositivos de pantalla reducida. Han creado un estándar en la resolución del sistema, de esta forma los desarrolladores saben que se van a encontrar, independientemente del dispositivo bajo el que se ejecute la aplicación. Microsoft provee de un entorno de trabajo a la altura de los mejores, con una gran cantidad de APIs que hacen que la programación resulte lo más sencilla posible.

Este sistema operativo esta siendo introducido en muchos modelos de diferentes marcas, cuyos dispositivos están a la altura de los mejores Android o iPhone. Microsoft ha entendido que es fundamental ofrecer una interfaz fluida y sencilla, y a partir de ahí, desarrollar el resto. Cumplida esta premisa podemos decir que los teléfonos que corren bajo Windows Phone tienen un rendimiento excelente.

La mayor desventaja viene como consecuencia de su mayor virtud. La llegada al mercado de forma tan tardía hace que a Windows Phone le resulte complicado competir con Apple iOS y Android, ya que estos tienen ganada una cuota de mercado muy amplia, y resulta muy difícil hacer cambiar de opinión a los usuarios, y más cuando no existen grandes saltos de calidad entre las alternativas. Como consecuencia de ello, el volumen de aplicaciones disponibles no es excesivamente grande, aunque si que hay que reconocer que se está haciendo un gran esfuerzo por parte de la comunidad y para el poco tiempo de vida del sistema se puede considerar que el catálogo es más que aceptable.

Por tanto, reúne las condiciones necesarias para ser una plataforma de desarrollo idónea, pero lamentablemente la cuota de mercado actual es insignificante, lo que hace que a los desarrolladores con pocos recursos les resulte una opción poco atractiva por el momento.

2. BlackBerry

Este tipo de dispositivos está claramente en decadencia. Hace unos cuanto años, eran el dispositivo ideal, ya que fueron los primeros en incorporar tecnologías de red para un uso permanente de ella. Sobrevivieron al inicio de los dispositivos táctiles y convivieron con ellos largo tiempo, pero a medida que la tecnología evolucionó, y las pantallas táctiles empezaron a tener una respuesta más precisa, provocando el auge de los dispositivos Android, hizo que la cuota de mercado de BlackBerry cayese precipitadamente.

Aunque siguen teniendo fieles seguidores, es una realidad que las ventas disminuyen año tras año. A pesar de ello, siguen sacando al mercado nuevos terminales, muchos de ellos de gama baja, tratando de ocupar cierta parte del sector, donde la lucha no está tan definida.

RIM, conscientes de la sangría que está ocurriendo, ha comenzado una transición iniciada con su tablet Playbook, cuyo sistema operativo QNX se espera que haga renacer a la compañía, implementándolo también en sus teléfonos móviles, migrando a las pantallas táctiles en lugar de los teclados físicos convencionales tan típicos de BlackBerry.

En cuanto a las aplicaciones, decir que al no contar muchos de sus terminales con capacidades multitáctiles se han quedado un tanto rezagados, y las posibilidades que ofrecen aplicaciones disponibles en otras plataformas se ven mermadas precisamente por este handicap. Muchos desarrolladores, al ver la decisión tomada por RIM, viendo la transición que se está llevando a cabo, creen que no merece la pena introducirse en la programación para un sistema operativo que tiene un fin más que próximo.

Quizás cuando QNX sea una realidad, las cosas pueden cambiar, pero con la rivalidad entre iOS y Android y la llegada de Windows Phone, parece complicado conseguir parte del pastel.

3. Android

Sin duda, a día de hoy, la gran alternativa a Apple iOS. Su virtud reside en que es un sistema abierto, con lo que su evolución es muy rápida y está siendo apoyado por la comunidad. Lleva muchos años en el mercado y tiene las bases bien asentadas. Al ser un sistema abierto puede ser implementado por cualquier dispositivo que lo soporte, este hecho es el que marca la principal diferencia entre el sistema operativo de Google y el de Apple.

Obviamente, tener a los fabricantes de tu lado es una ventaja, ya que en su momento, era la única alternativa libre a Apple. Al disponer de todo el soporte de la industria, la cuota de mercado se ha ido aumentando año a año, a igual que ha ocurrido con sus prestaciones. Esta es la tendencia que se ha ido produciendo a lo largo de los años y la que tiene vistas de continuar. De hecho, en la actualidad, Android ya copa más del 50% del mercado de los smart phones, por lo que a la hora de decidir una plataforma para desarrollar haya que tener en cuenta este dato, ya que cuantos más dispositivos existan con ese sistema operativo, más potenciales clientes de tu aplicación habrán.

El hecho de que Android esté abierto a todas las plataformas tiene también algunas desventajas.

Los dispositivos han ido evolucionando muy rápidamente, en cierto modo, esto puede ser un problema, ya que los sistemas operativos han de ser capaces de sacar el máximo rendimiento a los terminales más potentes, pero también tienen que garantizar el correcto funcionamiento en aquellos terminales más antiguos. Por tanto, existe un compromiso entre la compatibilidad y el rendimiento. Por ello, se producen numerosas actualizaciones de Android cada año, siendo el ciclo de vanguardismo de cada terminal bastante bajo.

Otro problema que existe con la diversidad de terminales, es la resolución de pantalla. Hoy en día la tendencia es que los dispositivos tengan cada día la pantalla más grande, y en la mayoría de los casos, eso implica una mayor resolución. Con lo que resulta complicado programar para distintas resoluciones. Si a este problema le unimos la diferencia de rendimiento entre unos terminales y otros, hace que garantizar el rendimiento óptimo y/o una visualización correcta de las aplicaciones en varios dispositivos sea una tarea complicada.

Existen otros factores determinantes que han hecho que la gente de Mountain View no les haya ganado la partida a Apple hace tiempo, como pueden ser la suavidad en la interfaz, la respuesta de la pantalla y la calidad de los terminales.

Este tercer punto se ha visto neutralizado a día de hoy, donde los terminales Android actuales están a la altura del iPhone correspondiente. Pero esto no siempre ha sido así. Hasta la llegada de terminales como el Samsung Galaxy S, siempre han estado un paso por detrás respecto al hardware o al consumo de batería, aunque esto es algo ajeno a Google, si que ha tenido sus consecuencias en la implantación del sistema.

En cuanto a las aplicaciones, existen una gran cantidad de aplicaciones en el Android Market, pero al igual que en otros aspectos, han ido más rezagados que Apple, el ecosistema económico creado por Apple en su App Store está más consolidado que Android Market, donde la mayoría de las aplicaciones existentes son gratuitas, y aquellas que son de pago, tienen un índice de venta mucho menor que en la plataforma rival. A pesar de ello, Android

Market está creciendo con mayor velocidad, y parece cuestión de tiempo que la calidad y número de aplicaciones se iguale, y que finalmente Android se imponga en este aspecto.

Aunque como ya he comentado antes, la cuota de mercado de Android está por encima del 50%, la mayor parte de ellos son terminales de gama baja, los cuales tienen un tamaño de pantalla menor y disponen de menos recursos. Por lo que el porcentaje real de terminales capaces de ejecutar aplicaciones que implementen las últimas tecnologías es menor. También el perfil del usuario tiene su repercusión. Existen estudios que documentan que los usuarios de Android descargan menos aplicaciones, y que la frecuencia de uso es mucho menor que en iOS.

En resumen, todo apunta a que Android en un futuro no muy lejano se impondrá. Las ventajas de ser un sistema abierto parecen mayores que los inconvenientes, y un factor generalmente determinante es el apoyo de la comunidad, y en este caso lo tiene. Sin embargo, los niveles de usabilidad y el ecosistema de aplicaciones creado por Apple están por encima de las capacidades actuales de Android.

Existen otras alternativas que descartamos inmediatamente ya sea por su baja cuota de mercado o por sus limitaciones técnicas como son: Bada, Meego, Symbian u otros.

4. Apple iOS

Finalmente nos decantamos por los dispositivos basados en Apple iOS, principalmente debido a que son actualmente los dispositivos que mayor partido sacan a las aplicaciones, ya sea por su usabilidad, el nivel de implantación de la AppStore, o por el tipo de usuario final.

Los elementos diferenciales que han hecho que Apple iOS se encuentre en una posición privilegiada son:

- **Apple crea su propio hardware**

  La ventaja de crear tu propio hardware es que puedes diseñar el sistema operativo a medida de los recursos de los que dispones, sin problemas de compatibilidad y sacando el máximo partido a los terminales. Como inconveniente está que el ritmo de actualizaciones de hardware es más lento, mientras que otras compañías dispondrán de terminales más avanzados a mitad del ciclo de vida del terminal de Apple.

- **Ecosistema de aplicaciones**

  Apple al haber creado su propio hardware y software tiene el control de la plataforma al completo. Al ser conscientes del potencial de su terminal, permitieron la creación de software por parte de terceros, y se cercioraron de crear un ecosistema controlado por ellos, capaz de hacer más grande su sistema operativo. Todo ello apoyado por un SDK muy cuidado. La respuesta de los desarrolladores fue inmediata, y la progresión de este ecosistema ha crecido año a año y más aún con la llegada del iPad.

- **Experiencia de usuario**

  A pesar de haber sido los que crearon el concepto de smartphone multáctil, la experiencia de usuario que ofrecen los terminales Apple sigue siendo la mejor de la industria. Apple,

como parte de su filosofía en todos sus productos, entiende que la experiencia de usuario ha de ser primordial, cuidando los detalles al máximo y haciéndolo de forma sencilla para el usuario.

Estos son algunos de los elementos diferenciales que hacen que Apple iOS sea una buena plataforma para la creación de aplicaciones.

Una vez elegida la plataforma de desarrollo, encontramos tres dispositivos para trabajar sobre ella, iPhone, iPod Touch y iPad. Los dos primeros son iguales a todos los niveles, por lo que generalmente no se hace referencia al iPod Touch ya que se da por sentado que se trata del mismo tipo de dispositivo. La única diferencia reside en la falta de algunos componentes en el iPod Touch como puede ser el GPS o la capacidad de realizar llamadas.

En cuanto al iPad, ofrece los mismos recursos que el iPhone, pero al disponer de una pantalla mayor, existe la posibilidad de personalizar las vistas para aprovechar al máximo las capacidades de una pantalla de mayores dimensiones.

En la mayoría de los casos, la opción de poder reorganizar los elementos de la aplicación de forma distinta condiciona ampliamente la aplicación y su propósito. El tamaño del dispositivo, de igual forma, condiciona la aplicación ya que los niveles de movilidad o los casos de uso son distintos.

Aunque el iPad ofrezca las mismas capacidades que el iPhone, como puede ser la localización, pensamos que el propósito del dispositivo es distinto aunque no por ello incompatible.

Por lo que concluimos que el iPad es indicado para ésta aplicación siempre y cuando la versión de iPad tuviese una visión más de documentación que de consulta como la versión de iPhone. Como el propósito es ofrecer información localizada, pensamos que adaptar la aplicación al iPad rompe con la filosofía de la aplicación, y sería más adecuado realizar una aplicación específica para iPad desde cero, adaptando las bondades y capacidades del iPad en lugar de portar la versión de iPhone.

De todas formas, la aplicación se ejecuta sin problemas en el iPad, con un funcionamiento exacto a la versión de iPhone, simplemente no se aprovecha la capacidad de una pantalla de mayor tamaño.

Por tanto, si nos centramos de forma específica en el iPhone, podemos recalcar algunas de sus características más importantes, que hacen de éste dispositivo una plataforma perfecta para la implementación de aplicaciones de estas características.

iPhone

Dispositivo precursor de la tecnología móvil actual, basa su funcionamiento en 4 puntos principales:

1. Pantalla

La pantalla del iPhone es de 320x480 píxeles en todas las versiones de iPhone y iPod Touch excepto en el iPhone 4, iPhone 4S y el iPod Touch de segunda generación donde esta

resolución aumenta a 640x960 píxeles, que es justamente el doble, ya que así se pueden escalar las aplicaciones de una forma sencilla de un dispositivo a otro. El tamaño de la pantalla se mantiene en 3,5 pulgadas para todos los dispositivos y ha pasado a ser una referencia en la industria de los teléfonos inteligentes.

Si comparamos este tamaño de pantalla con el tamaño de pantalla de los ordenadores de sobremesa o portátiles vemos que se ofrece un espacio bastante limitado para presentar la interfaz y el contenido de nuestras aplicaciones. Las aplicaciones de iPhone también eliminan la noción de las ventanas múltiples. Sólo dispondremos de una ventana con la que trabajar. Podemos cambiar el contenido dentro de dicha ventana, pero los conceptos de escritorio y de las aplicaciones con varias ventanas, han desaparecido.

Los límites de la pantalla no suponen un problema. Las herramientas de desarrollo para iPhone nos ofrecen la posibilidad de crear aplicaciones tan completas como su software de escritorio. No obstante, la visualización de las mimas presentará un diseño de la interfaz más estructurado y eficiente.

Los gráficos que mostremos en la pantalla pueden incluir complejas visualizaciones animadas en 2D y 3D gracias a la implementación OpenGL ES disponible en todos los modelos de iPhone. OpenGL es un estándar industrial orientado a la definición y manipulación de imágenes, ampliamente utilizado para la creación de juegos. Las versiones posteriores de iPhone, comenzando en el iPhone 3GS, mejora dichas capacidades mediante un chipset 3D actualizado y una versión más avanzada de OpenGL, pero todos los modelos cuentan con un tratamiento de gráficos más que respetable.

2. Limitaciones en los recursos de aplicaciones

De igual modo que en las pantallas de alta definición de nuestros ordenadores de sobremesa y portátiles, estamos acostumbrados a los procesadores que pueden trabajar más rápidamente que el tiempo que nos lleva hacer un clic. El iPhone emplea un ARM 412 MHz en los primeros modelos, una versión 600 MHz en el 3GS, 800MHz en el iPhone 4 y 800MHz de doble núcleo en el iPhone 4S, todos ellos underclocked, es decir, con la velocidad del reloj reducida, con el fin de conservar la vida útil de la batería.

Apple se ha esforzado mucho para hacer que el iPhone responda correctamente sin importar lo que estamos haciendo. Inicialmente el iPhone OS, ahora renombrado a iOS, era capaz de ejecutar únicamente una aplicación al mismo tiempo. Desde la aparición de iOS 4, esta limitación desapareció, aunque las dos primeras generaciones del dispositivo no soportan más allá de la versión iPhone OS 3, por lo que esta limitación sigue vigente en estos dispositivos.

Otra limitación que tenemos que tener en cuenta es la memoria disponible. En las dos primeras generaciones del dispositivo se contaba con 128MB de RAM disponibles para todo el sistema. No existe memoria virtual, por lo que será necesario manejar cuidadosamente los objetos que cree nuestra aplicación. El iPhone 3GS cuenta con 256 MB de RAM, mientras que en el iPhone 4 esta cantidad aumenta hasta los 512 MB, en el caso del iPhone 4S se especula que la cantidad de memoria se mantiene respecto a su predecesor.

3. Conectividad

La conectividad es una de las áreas en las que iPhone realmente brilla. A diferencia de los portátiles actuales, el iPhone cuenta con la capacidad de estar conectado permanentemente a través de un proveedor móvil. Cada versión más reciente de iPhone mejora estas capacidades.

El acceso a Internet vía móvil se completa con las tecnologías WiFi y Bluetooth integradas en el dispositivo. WiFi puede ofrecer velocidades similares a las de un ordenador de sobremesa siempre y cuando tengamos una red inalámbrica al alcance. Por otro lado, Bluetooth puede usarse para conectar múltiples dispositivos a nuestro teléfono.

En el caso de los desarrolladores, se puede hacer uso de la conexión permanente a Internet para actualizar el contenido de su aplicación, mostrar páginas Web o crear juegos para varios jugadores.

4. Entrada y retroalimentación

El iPhone es especialmente brillante cuando hablamos de los mecanismos de entrada y retroalimentación, y de las capacidades que tenemos para trabajar con ellos. Podemos leer los valores de entrada desde la pantalla multi-táctil, captar el movimiento y la inclinación a través del acelerómetro, determinar dónde se está utilizando el GPS, reconocer en que posición estamos mirando el dispositivo gracias a la brújula digital, y saber el uso que se está haciendo del iPhone con los sensores de proximidad y de luz.

El teléfono en sí, puede ofrecernos tanta información útil para nuestra aplicación sobre cómo y dónde se está siendo utilizado, que el propio dispositivo se convierte en un auténtico controlador de movimientos.

El iPhone también es compatible con la captura de imágenes y vídeo directamente a nuestras aplicaciones, abriendo un auténtico abanico de posibilidades para que interactuemos con el mundo real. Ya existen aplicaciones que identifican objetos que aparecen en las fotos tomadas y encuentran referencias sobre ellos en Internet.

Por último, para cada acción que el usuario llevo a cabo al interactuar con nuestra aplicación, podemos proporcionar retroalimentación. Es decir, cualquier tipo de información visible en pantalla, o un sonido de alta calidad o una vibración del dispositivo.

Nunca antes ha existido un dispositivo que ofrezca tantas posibilidades a un desarrollador.

Ahora vamos a revisar el Entorno de Trabajo (IDE), el Software Development Kit (SDK) y el lenguaje de programación que se utiliza en él.

‣ Xcode

Xcode es un entorno de trabajo diseñado por Apple. Dicho entorno ofrece todas las herramientas necesarias para programar tanto para Mac OS X como iOS.

- **Xcode IDE**

Xcode está altamente integrado con los frameworks Cocoa y Cocoa Touch, creando un entorno de desarrollo productivo y fácil de usar, y que es los suficientemente potente para que sea esta misma herramienta la que ha usado Apple para la creación de Mac OS X y iOS.

Diseñado desde cero para aprovechar las más modernas tecnologías ofrecidas por Apple, Xcode integra todas la herramientas que un desarrollador necesita. Una única interfaz que permite suaves transiciones entre el editor de codigo fuente, el depurador y el diseño de interfaces, todo ello en la misma ventana.

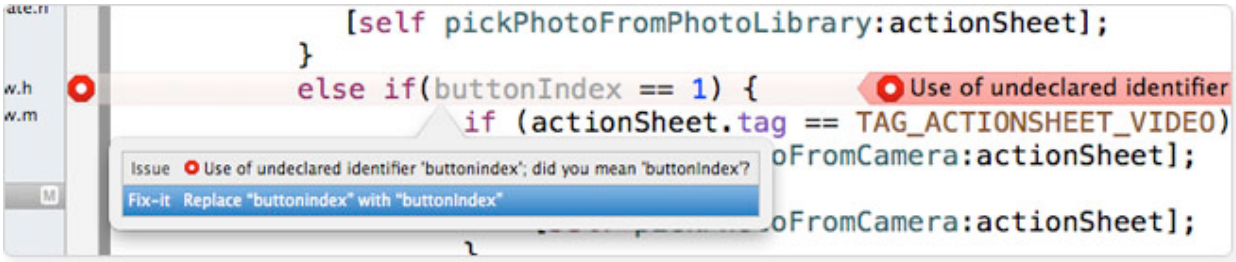
El área de trabajo de Xcode se centra en el rendimiento. Mientras escribes, Live Issues te avisa inmediatamente de los posibles errores de código, mostrando un mensaje con mayor detalle. Pulsa el botón de Run para ejecutar tu aplicación, o subir el contenido de la aplicación a tu dispositivo para empezar inmediatamente con la depuración. Pasa el puntero del ratón sobre una variable para inspeccionar su valor en tiempo de ejecución, sin dejar en ningún caso el editor.

Plena interacción con el portal web de desarrolladores, Xcode suministra nuevos dispositivos iOS con un solo click, puedes almacenar y firmar de forma segura tus aplicaciones para Mac e iOS y enviarlas directamente a la App Store.

- **Apple LLVM Compiler**

Es el compilador que proporciona Apple en su Xcode. Apple LLVM Compiler hace mucho más que compilar tu aplicación. El mismo analizador de código que utiliza Xcode a la hora de compilar código tanto en C como en C++ y Objective-C, ahora también se utiliza para

completar código en tiempo real. Mientras escribes, Apple LLVM Compiler está, de forma constante, evaluando lo que escribes, identificando errores mostrados como Live Issues, y proporcionando posibles soluciones por ti. Otros compiladores pueden decirte que está mal, Apple LLVM lo corrige por ti.



- **Instruments for Perfomance and Behavior Analysis**

Las aplicaciones Mac OS X o iOS ofrecen una gran experiencia de usuario, y eso significa mucho más que un diseño intuitivo o una buena presentación. Las buenas aplicaciones deben ser rápidas, fluidas y que respondan de forma veloz. Para ayudar a ello, Xcode proporciona una herramienta llamada Instruments, una aplicación única que ayudar a localizar los cuellos de botella de rendimiento en Mac OS X y aplicaciones iOS.

Instruments recopila en tiempo real tanto datos, como memoria o tiempo de uso de la CPU, incluso en un Mac conectado de forma remota a un iPhone. Los datos recopilados se muestran gráficamente como pistas en el tiempo, haciendo fácil la detección de problemas, e identificando de forma sencilla las lineas de código implicadas.

Instruments ayuda a garantizar que una maravillosa interfaz sea acompañada por una respuesta sublime.

- **iOS Simulator**

El simulador iOS ejecuta tu aplicación de la misma forma que lo hace un dispositivo iOS. Al ser más rápido para ejecutar y depurar, el simulador iOS es perfecto para comprobar que todo funciona como el desarrollador espera. Se puede simular incluso la orientación del dispositivo o gestos multi táctiles.

Apple iOS

IPhone OS (ahora conocido como iOS), es el Sistema Operativo utilizado por iPod touch, iPhone y iPad desarrollado por Apple. Es una variante del Mac OS X, lo que significa que usa un kernel XNU basado en Mach, hereda parte de las tecnologías desarrolladas por NeXT y utiliza frameworks Cocoa entre otras cosas que se detallarán a continuación.

Tanto Mac OS X como iOS, tienen cuatro capas de abstracción, la diferencia entre ambos recae en la última de estas:

1. **Capa del Núcleo del Sistema Operativo (Core OS):** realiza la gestión de controladores, memoria virtual, sistema de ficheros, TCP/IP, sockets, seguridad, gestión de memoria y comunicación entre procesos entre otras funciones.

2. **Capa de Servicios principales (Core Services):** esta capa proporciona los servicios fundamentales del sistema, que todas las aplicaciones van a usar (directa o indirectamente). Permite realzar conexiones a la red, acceso a ficheros, acceso a la agenda, usar la base de datos SQLite, ubicación del dispositivo y gestión de threads entre otras funciones.

3. **Capa de Medios de Comunicación (Media):** el conjunto de Frameworks y librerías que forman esta capa permiten construir aplicaciones con gráficos avanzados, reproducción de video, audio, animaciones o imágenes.

4. **Capa de Cocoa (para Mac OS X) - Capa Cocoa Touch (para iOS):** El conjunto básico de herramientas que permiten crear y acceder a los objetos y estructuras de datos básicos, creación de interfaces de usuario, conectar la interfaz con controladores para manejar eventos, etc.

El lenguaje de programación utilizado para el desarrollo de aplicaciones tanto en iOS como en Mac OS X es Objective C.

Objective-C

Objective-C fue creado en el año 1980 y se trata de una extensión del lenguaje C. Añade muchas características adicionales a C y lo que es más importante, una estructura POO. Objective-C utiliza principalmente para el desarrollo de aplicaciones de Mac OS X y iOS, lo cual sedujo a un entregado grupo de seguidores que aprecian sus capacidades y sintaxis.

Las sentencias de Objective-C son más fáciles de leer que las de otros lenguajes de programación y suelen descodificarse con un simple vistazo. Por ejemplo, considere la siguiente línea que compara si el contenido de la variable llamada myName es igual a John:

```
[myName isEqualToString:@”John”]
```

No requiere mucho esfuerzo mental ver lo que ocurre en ese fragmento de código. En C tradicional, la misma sentencia de código se escribiría de la forma siguiente:

```
strcmp(myName,“John”)
```

La sentencia en C es algo más corta, pero es difícil imaginarse lo que hace el código sin saberlo.

Debido a que Objective-C está implementado como una capa situada por encima de C, sigue siendo completamente compatible con código totalmente escrito en C.

Model-View-Controller

Es un patrón de diseño de aplicaciones que define una separación clara entre los componentes críticos de nuestras aplicaciones. Como su nombre indica, MVC define tres partes de una aplicación.

- Un modelo proporciona los datos internos y los métodos que ofrecen información al resto de la aplicación. El modelo no define la apariencia o el comportamiento de la aplicación.
- Una o más vistas construyen la interfaz de usuario. Una vista está compuesta por diferentes widgets de pantalla (botones, campos, interruptores, etc.) con los que interactúa el usuario.
- Un controlador suele estar vinculado a una vista. El controlador es responsable de recibir la entrada del usuario y reaccionar en consecuencia. Los controladores pueden acceder y actualizar una vista usando información del modelo, así como actualizar el modelo empleando el resultado de las interacciones del usuario en la vista. Resumiendo, enlaza los componentes de MVC.

Modelo de datos

Al tratarse de una aplicación cuya finalidad es la de mostrar información al usuario, existen muchas posibilidades a la hora de almacenar y recuperar dicha información.

- **Modelo local:**

Podemos almacenar la información en el interior de la aplicación, ya sea a través de bases de datos internas, diccionarios, arrays o con la creación de objetos de forma estática.

- **Modelo remoto:**

Podemos almacenar la información en un servidor remoto, ya sea en una base de datos o un fichero XML, y leer esa información mediante consultas a la base de datos o mediante parsing del XML.

Ambos métodos tienen sus ventajas e inconvenientes, y elegir el modelo apropiado depende del tipo de aplicación y del uso que se le vaya a dar.

Almacenar la información de forma interna tiene como principal ventaja la velocidad de procesado y la ausencia de conectividad de red. Como inconvenientes, el volumen de la aplicación aumenta de manera considerable, si la información es cambiante existirá la necesidad de actualizar la aplicación de forma constante, si dicha información es actualizable por los propios usuarios entonces este modelo no resulta viable.

La consulta de información de forma remota hace que al no existir una dependencia con la aplicación, los datos pueden ser actualizados constantemente de forma totalmente transparente para el usuario, sin necesidad de cambiar la estructura de la aplicación y por consiguiente, sin tener que pasar por la App Store para su aprobación y el retardo que ello implica. Esto precisa de una actividad de red constante, con la pérdida de rendimiento que esto conlleva.

Por tanto, determinar que modelo es el más apropiado dependerá de la naturaleza de los datos con los que vayamos a trabajar.

Realidad Aumentada

La realidad aumentada consiste en la utilización combinada de diferentes recursos de los que disponen los dispositivos móviles para proporcionar información localizada sobre una captura a tiempo real de la cámara de dicho dispositivo.

De forma que si se combinan los datos proporcionados por los acelerómetros, la brújula, el gps, y la cámara, en combinación con la conectividad a internet, podemos mostrar información localizada en tiempo y espacio sobre la vista de la cámara.

Apple no proporciona ningún API en su SDK que de soporte a la realidad aumentada, de forma en que si el desarrollador está interesado en implementar una aplicación que tenga como funcionalidad la realidad aumentada, tendrá que coordinar y gestionar por si mismo los recursos de los que dispone el dispositivo para reproducir de la forma más fidedigna la realidad aumentada.

Ahora bien, existen frameworks, algunos de ellos libres, otros cerrados pero gratuitos y otros de pago, que proporcionan en mayor o menor medida el código necesario para la implementación de la realidad aumentada.

He analizado algunas de las propuestas existentes como son Layar, QCAR del Qualcomm, 3DAR y ARKit.

Layar es un framework desarrollado por la compañía homónima que se dedica específicamente en la creación de aplicaciones y servicios relacionados con la realidad aumentada.

El fundamento de layar reside en la sinérgia con la aplicación Layar existente para diferentes plataformas móviles, de forma en que los desarrolladores pueden crear diferentes capas (layers) las cuales son almacenadas en una base de datos a cargo del desarrollador, mientras que Layar dispone de un servicio web que es capaz de recibir una respuesta JSON mediante la consulta a la base de datos. Por tanto, la aplicación dispone de la capacidad de mostrar las capas generadas por los diferentes desarrolladores mediante el parsing del JSON.

Desde hace unos seis meses, Layar proporciona un framework llamado Layar Player, el cual puede ser implementado por los desarrolladores en sus aplicaciones, de tal forma en que se crea una instancia de Layar dentro de la propia aplicación que lee las capas generadas por dicho desarrollador.

Esta solución conviene tenerla en cuenta porque el desarrollador simplemente tiene que importar un framework y crear una instancia de Layar que apenas son 3 líneas de código. Para el desarrollo de la capa, existe una gran documentación paso por paso de cual es la estructura que debe tener la base de datos. Además, es una solución gratuita, y tiene un acabado muy profesional.

En cambio tiene algunos inconvenientes, como por ejemplo, la personalización es limitada, es decir, Layar a través de su servicio web ofrece diversas opciones de visualización, pero estamos restringidas a ellas, además de incorporar un logo. La instancia de Layar dispone únicamente de la funcionalidad que Layar le ha dado, es decir, no podemos incorporar nuevas funcionalidades ya que no disponemos de acceso al código. Además, al incorporar un elemento externo, hace que nuestra aplicación esté más expuesta, de forma en que si los servicios que Layar ofrece no estuviesen disponibles, no podemos hacer nada por remediarlo.

Otro problema que he encontrado y que ha sido determinante para descartar esta opción, ha sido que con la actualización de iOS a la versión 5, el Layar Player actual ha dejado de funcionar correctamente, y no hay una fecha definida para la publicación de una nueva versión compatible. Lo que me ha hecho mirar otras opciones.

A parte de Layar, existen otras empresas que ofrecen soluciones basadas en la misma filosofía como puede ser QCAR de Qualcomm. Sin llegar a implementarlo, he descartado esta opción por estar más enfocado a la reproducción de objetos en 3D.

Haciendo una búsqueda más exhaustiva, he encontrado otras soluciones que se aproximan más a los requisitos de nuestra aplicación.

3DAR ofrece un framework que no es más que una subclase de MKMapView con muchos añadidos. La ventaja de este framework es la compatibilidad con las últimas versiones del sistema operativo, se percibe una alta actividad en cuanto a las actualizaciones. Por contra, el rendimiento que ofrece la vista de mapa es muy pobre, y además es necesario una licencia en

caso de que la aplicación vaya a ser comercializada, ya que se valida la aplicación en sus servidores cada vez que es ejecutada.

Finalmente he optado por una solución abierta creada por la comunidad, que aunque sus prestaciones son menores, existe la posibilidad de acceder al código fuente y modificarlo en función de las necesidades de la aplicación. Además, ofrece un mejor rendimiento y no depende de terceros. En el capítulo siguiente se explica con más detalle el funcionamiento y los cambios realizados al código original.

Estadísticas

Una vez completada la aplicación conviene hacer un seguimiento del uso que los usuarios hacen de la aplicación, ya que esto nos puede servir para identificar que secciones de la aplicación son las más utilizadas y cuales son las menos interesantes. De esta forma podemos conocer cuales son los aspectos a mejorar.

Además podemos conocer datos estadísticos que nos pueden ayudar a la hora de dirigir una campaña publicitaria o ampliar funcionalidades de la aplicación en futuras actualizaciones. Con el uso de estadísticas somos capaces de identificar los usuarios por edades o por nacionalidades. También podemos realizar un seguimiento de las descargas diarias o calcular el tiempo medio de navegación, y todo ello representado mediante gráficas o tablas personalizables.

Apple ofrece este servicio dentro de su portal de ventas llamado iTunes Connect, al que se puede acceder a través de la pagina web correspondiente y también a través de la aplicación específica para ello. Su funcionalidad se reduce a meras estadísticas de venta. Por ello, existen otros productos que completan la oferta ofreciendo los servicios carentes en iTunes Connect.

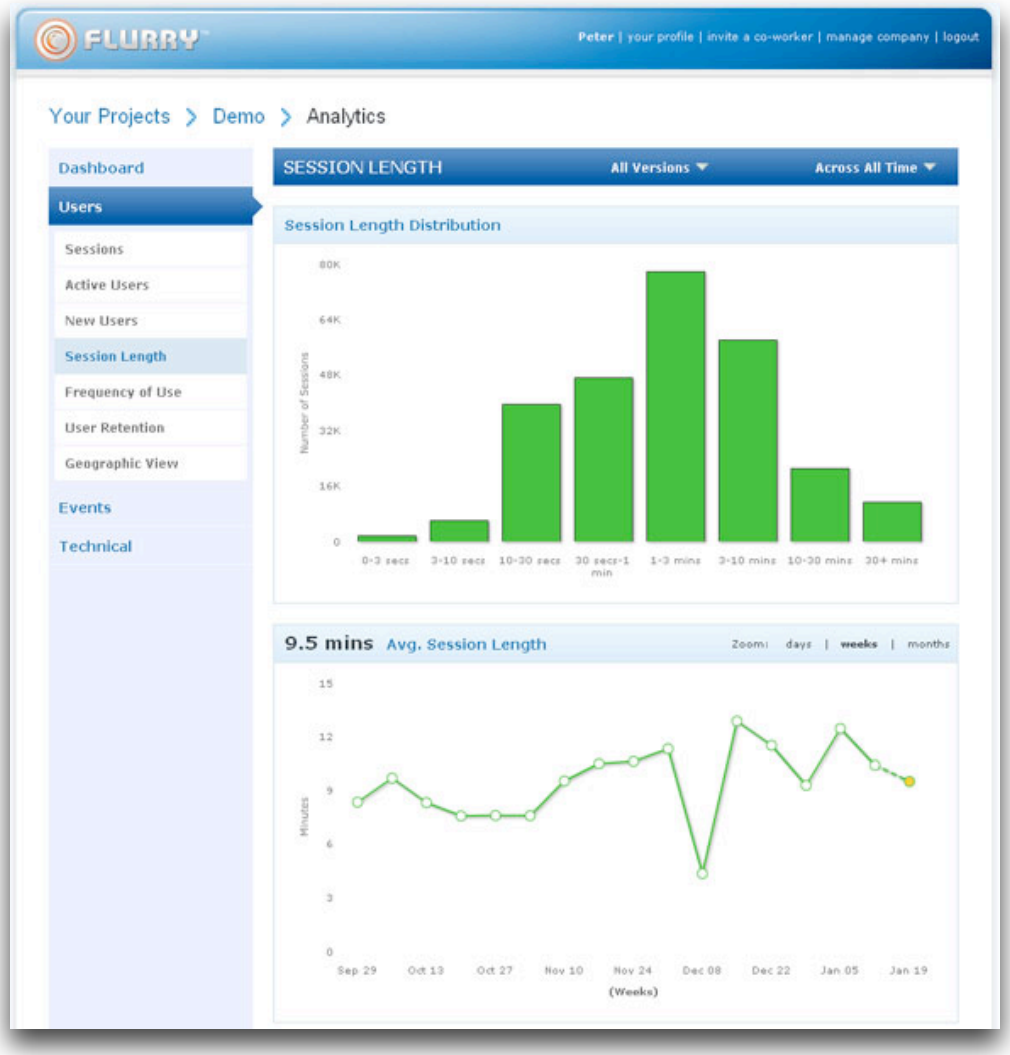
Google ofrece este tipo de servicio a través del tradicional Google Analytics. Google dispone de un SDK específico para dispositivos iOS, de forma que resulta sencilla la implementación y es capaz de proporcionar datos que el sistema de Apple no proporciona. A nivel interfaz resulta prácticamente igual a los servicios de análisis que ofrece google para páginas webs.

Existen otras soluciones como Flurry, que ofrece los mismos servicios que Google Analytics pero con una implementación mucho más sencilla.

Con Flurry podemos identificar diferentes propiedades del usuario como la edad, o la localización, dotando a las estadísticas de una mayor profundidad respecto a iTunes Connect.

También existe la posibilidad de contabilizar la duración de las sesiones, si a esto le unimos que podemos determinar a que partes de la aplicación se accede, seremos capaces de averiguar con un alto grado de fiabilidad cuales son las vistas más interesantes para el usuario. De esta forma podemos saber cuales son los aspectos a mejorar o cuales podrían ser las vista más adecuadas para introducir publicidad.

Otros aspectos que Flurry es capaz de monitorizar son: el modelo del dispositivo, el idioma o el género.



3. Arquitectura de la solución

Puesta a punto

El primer paso para introducirse en el desarrollo de aplicaciones para dispositivos iOS es descargarse el SDK que se encuentra disponible en el App Store de forma gratuita, o en caso de estar utilizando una versión anterior a Mac OS X Lion, en el DVD o USB de instalación.

El SDK proporciona todos los elementos necesarios para la programación de aplicaciones para dispositivos iOS. Una vez instalado el SDK, el siguiente paso es darse de alta en el Apple Developer Program, esta subscripción no tiene asociada ninguna cuota de registro, en cambio existen ciertas limitaciones.

En caso de querer descargar las versiones beta de iOS, o tener la posibilidad de cargar las aplicaciones que programemos en un dispositivo iOS físico o distribuirlas en el App Store, necesitaremos pagar por alguno de los programas de pago que ofrece Apple.

- El programa estándar cuesta 79 € al año y permite probar las aplicaciones en dispositivos físicos, hasta 100 dispositivos, y también publicar nuestras aplicaciones en el App Store.

- Existe otro programa empresarial que cuesta 299 € el año que permite distribuir las aplicaciones internamente en lugar de hacerlo a través del App Store.

En nuestro caso, he optado por adquirir el programa estándar, ya que me permite poder probar la aplicación en mi propio teléfono y distribuirla en el App Store en un futuro.

De hecho, existen ciertos recursos que no son posibles testearlos a través del simulador que ofrece el SDK, como por ejemplo, la cámara o ciertos sensores.

La universidad dispone de una licencia, la cual me hubiese permitido probar la aplicación en un dispositivo físico sin tener que pagar por mi cuenta la licencia estándar. En cambio, en el momento en que decidiese publicar la aplicación, hubiese tenido que comprar la licencia igualmente.

El proceso de adquisición de alguna de las licencias que Apple ofrece está bien documentado en la página de desarrolladores de Apple: developer.apple.com.

Después del registro gratuito y de asociar dicha cuenta a la cuenta de iTunes Store, hace falta descargar un perfil de suministro para el desarrollo (Development Provisioning Profile) e instalarlo en el sistema. Siguiendo los pasos que documenta la web, tras crear un identificador de la aplicación, generar unos certificados y agregar correctamente los dispositivos, ya estaremos listos para la carga de nuestra aplicación en nuestros dispositivos.

Especificaciones funcionales

Como bien se ha expuesto anteriormente, la aplicación está enfocada al sector turístico y el propósito de ésta es proporcionar la información necesaria al cliente. Esta información será representada de diferentes formas, ya sea en un mapa, en una tabla o en una vista web.

Para una representación de la información lo mas intuitiva posible, se ha desarrollado la aplicación bajo la estructura de barra de pestañas (tab bar). Este será el modelo predeterminado de navegación, aunque como veremos, bajo este modelo se anidarán otros métodos de navegación.

Por tanto la aplicación dispondrá de la siguiente estructura:

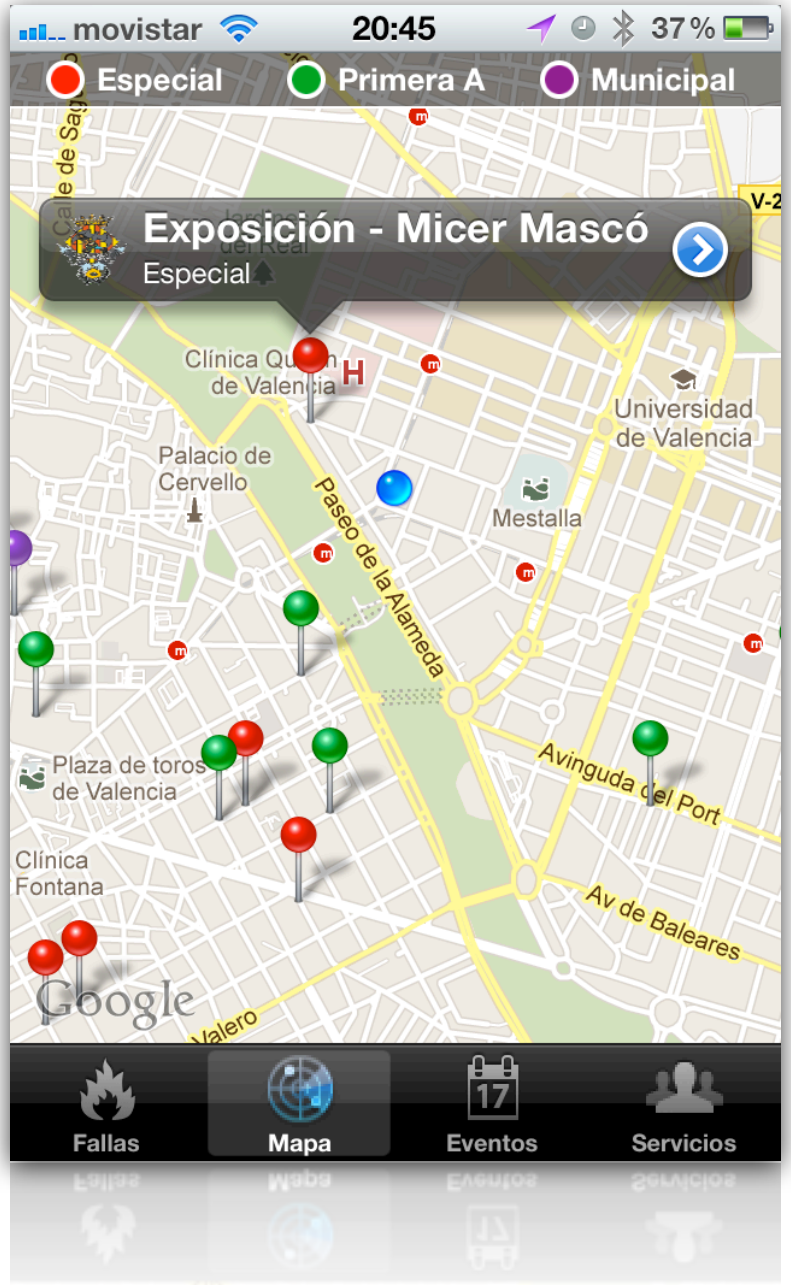
La primera pestaña, **Fallas**, es la vista que se muestra por defecto al ejecutar la aplicación. Muestra un listado con todas las fallas y las ordena por distancia, mostrando la primera la falla que más próxima se encuentre al dispositivo. Existe la posibilidad de ordenar las fallas por categoría en lugar de por distancia.

Si pulsamos sobre una de las fallas, aparece un menú emergente con dos posibilidades, mostrar una ficha con información detalla de la falla en cuestión, o mostrar en la aplicación Mapas del dispositivo la ruta desde el punto en el que nos encontramos hasta la falla objetivo, ya sea en coche, en transporte público, o andando.

En la ficha específica de cada falla se muestra en una vista adaptada para la navegación en dispositivos móviles toda la información que necesita el usuario. La información que se muestra es la siguiente:

Escudo de falla, nombre de la falla, calle, año de creación, premio 2012 de la falla, premio 2011 de la falla, presupuesto de la falla, boceto de la falla, descripción del boceto de la falla, foto de la fallera mayor, nombre de la fallera mayor, nombre del presidente, nombre del artista fallero, premio 2012 de la falla infantil, premio 2011 de la falla infantil, presupuesto de la falla infantil, boceto de la falla infantil, descripción del boceto de la falla infantil, foto de la fallera mayor infantil, nombre de la fallera mayor infantil, nombre del presidente infantil y nombre del artista fallero de la falla infantil.

La siguiente pestaña de la barra de pestañas llamada **Mapa**, muestra un mapa con la posición actual del dispositivo en el centro de la pantalla. Alrededor una serie de pins que representan las fallas que se encuentran próximas a nuestra localización.

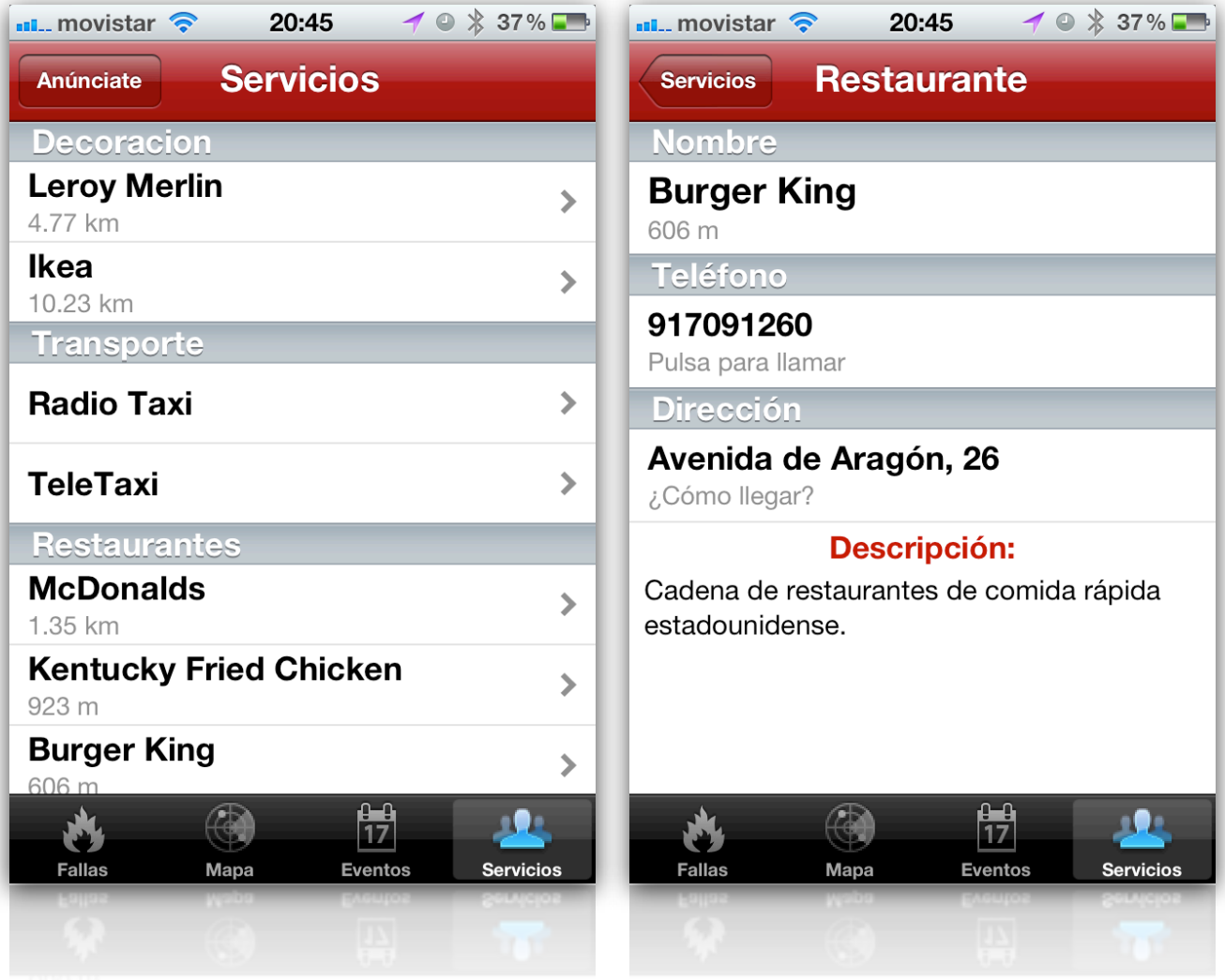


Como bien se indica en la leyenda, los pines rojos representan las fallas de especial, mientras que los pines verdes representan a las fallas de primera A, existe un pin morado que representa la falla del Ayuntamiento.

La tercera pestaña, **Eventos**, muestra un listado con los eventos de fallas agrupados por días. Cada evento tendrá una ficha donde se detalla con mayor exactitud el contenido del evento.

En la ficha descriptiva se especifica el día, la hora, el lugar y una descripción detallada del evento en cuestión.

La cuarta pestaña, **Servicios**, muestra un listado de los servicios ofertados, agrupados por el tipo de servicio, además se muestra bajo el nombre, la distancia a la que se encuentran, si pulsamos sobre uno de ellos, accederemos a una ficha más detallada del servicio.



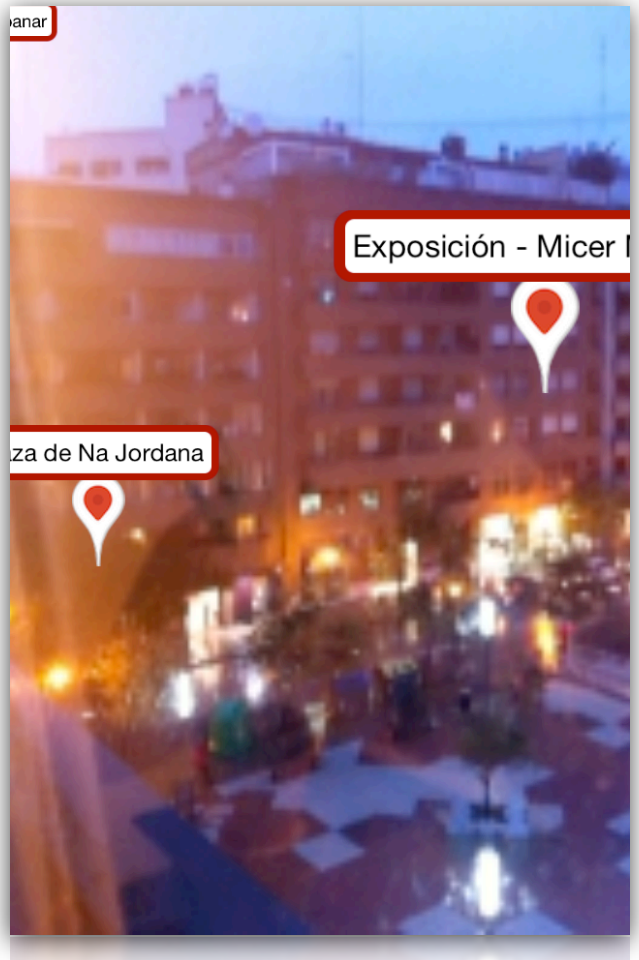
En la ficha específica de cada servicio se muestra el teléfono, la dirección y una breve descripción. Si pulsamos sobre el número de teléfono, el iPhone llamará directamente al número. Si pulsamos sobre la dirección, nos calculará la ruta desde la posición actual hasta el destino representado en una vista de mapa.

Además de estas cuatro pestañas, existen dos vistas a las que se accede de un modo distinto

Por un lado, la vista de **Noticias**, que muestra la última noticia publicada. Existe la posibilidad de notificar al usuario de la publicación de una nueva noticia mediante una notificación Push.

A través de las notificaciones Push se mantiene al usuario actualizado de forma instantánea de cualquier noticia importante que acontezca.

Y para finalizar, dentro de la vista **Mapa**, si ponemos el dispositivo en posición vertical, se activará la realidad aumentada, mostrando en pantalla las imágenes que están siendo capturadas por la cámara del dispositivo y la información de la falla sobre las imágenes, dando la sensación de que la información flota sobre el mundo real.



El tamaño del texto que aparece en pantalla aumentará o disminuirá en función de la distancia a la que se encuentre la falla. Si pulsamos sobre alguno de los paneles, aparecerá un nuevo panel donde se indica la distancia.

‣ Tecnología empleada

Una vez propuesta la estructura de la aplicación, vamos a ver cuales son las tecnologías necesarias para cumplimentar con todas las funcionalidades.

XML

Primero, tenemos que analizar la naturaleza de los datos que vamos a manejar. Los datos son principalmente información acerca de las fallas. En un principio, estos datos no van a ser cambiantes, por lo que parece que lo indicado sería alguna forma de almacenamiento local como podría ser CoreData, arrays o diccionarios. Ahora bien, como la aplicación está diseñada con la mente puesta en fallas, hace que cualquier cambio que sea necesario durante la semana fallera no se verá reflejado hasta una actualización. Por tanto, necesitamos primar la velocidad de actualización, antes que el rendimiento o la optimización del uso de la red. Por ello, se decide que los datos sean obtenidos mediante el parsing de XML, ya que la actualización de los datos se hace de forma totalmente transparente para el usuario.

Ahora bien, podemos optimizar este método, almacenando de forma local parte de la información que sepamos que tiene un carácter más estático y que supondría un consumo de ancho de banda excesivo. Por ello, hemos decidido almacenar los escudos de las comisiones falleras en el interior de la aplicación, con lo que se consigue minimizar los tiempos de carga y dotar a la aplicación de la máxima fluidez.

Notificaciones Push

Apple ofrece un servicio que consiste en la posibilidad de enviar notificaciones de aplicación al usuario aunque dicha aplicación no se esté ejecutando.

El sistema funciona de la siguiente forma:

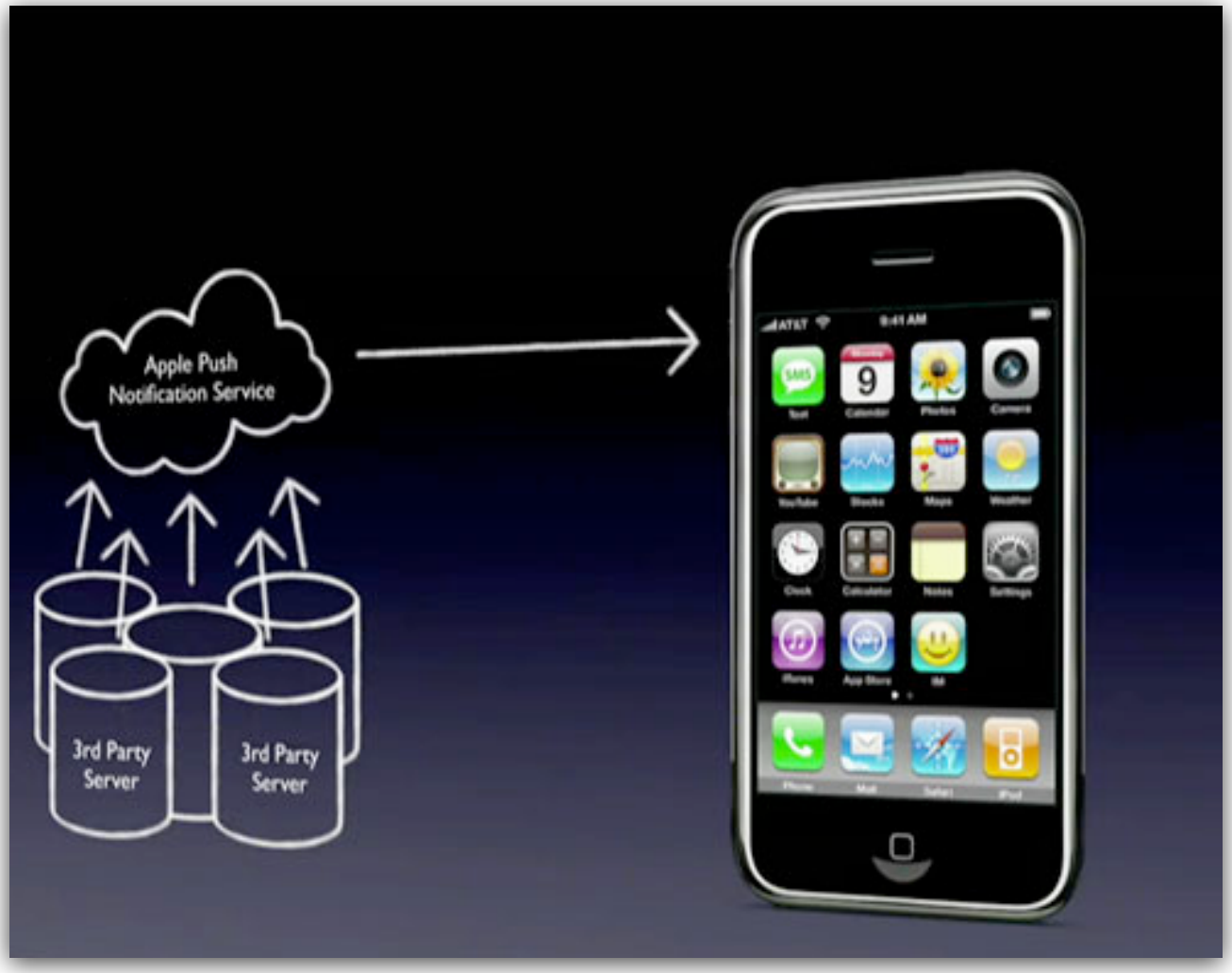
El desarrollador pone en funcionamiento un servidor que recibe las peticiones de los terminales. Este servidor almacena el identificador único de cada terminal, creando una base de datos con los terminales que tienen esa aplicación instalada.

El desarrollador deberá implementar el código necesario para que el dispositivo en cuestión envíe su identificador único al servidor correspondiente.

Una vez puesto en funcionamiento el servidor, debemos establecer comunicación con los servidores de notificaciones Push de Apple. Todo este proceso se realiza a través del portal de desarrolladores del que dispone Apple, mediante el intercambio de certificados y el uso de la licencia correspondiente.

Establecida ya la comunicación con los servidores de Apple, hemos de implementar un sistema para que nuestro servidor sea capaz de enviar las notificaciones a los servidores de Apple. Para ello Apple proporciona en su documentación la información necesaria para la creación de mensajes tipo JSON con el formato adecuado.

Todo este intercambio de información entre servidores tiene un motivo, el ahorro de energía. Con este fin, en lugar de que cada aplicación se conecte a su servidor correspondiente, Apple sólo permite que exista una conexión permanente a su propio servidor, de tal forma en que las notificaciones de todas las aplicaciones llegan a través de un único hilo de comunicación, con el ahorro de energía que ello conlleva.



Una vez entendido el funcionamiento, necesitamos de un servidor que realice la comunicación con los servidores de Apple.

En la mayoría de los casos, disponer de un servidor sólo para las notificaciones Push no resulta rentable ni eficiente. Por ello, existen compañías que ofrecen este servicio a través de internet, como por ejemplo, **Urban Airship**.

Mediante la implementación de su API en el código de las aplicaciones, podemos evitar el costo de un servidor y utilizar su servicio de forma gratuita, aunque también existe una opción de pago, la cual permite tener un mayor control sobre los dispositivos asociados. Este servicio dispone de una interfaz web que permite el envío de notificaciones. Existen también APIs para las diferentes plataformas móviles, pudiendo gestionar las notificaciones indistintamente de la plataforma.

Obviamente he optado por esta solución, que ofrece unos servicios superiores respecto a un servidor básico, siendo además, más barato, más seguro, y con más recursos que montar nuestro propio servidor.

Para la implementación, Urban Airship ofrece un API, el cual está muy documentado e indica paso por paso cual es el código a agregar y en que clases.

Realidad Aumentada

Queremos darle un toque diferencial a nuestra aplicación mediante la implementación de realidad aumentada.

Habiendo estudiado todas las posibilidades que se ofertan, nos hemos decantado por la implementación de **ARKit**, ya que es la opción que menos restricciones nos impone, principalmente porque es una solución abierta y tenemos acceso a todo el código.

ARKit es una librería open source de realidad aumentada creada por Zac White en Septiembre de 2009. Actualmente esta solución se encuentra obsoleta, pero a partir de ella han surgido distintas evoluciones unas mejores que otras. En nuestro caso vamos a utilizar la versión creada por Niels Hansen, que me ha parecido, de largo, la versión más estable y evolucionada.

El proyecto se encuentra hospedado en Github, donde además de la librería de ARKit se encuentra un proyecto de ejemplo donde se implementa la realidad aumentada.

El proceso de implementación ha consistido en examinar detenidamente el código de ejemplo e identificar como se han de llamar las clases necesarias. A partir de ahí, he ido modificando el código adaptándolo a las necesidades de la aplicación y tratando de respetar el estilo impuesto hasta ahora.

Estadísticas

Es conveniente la utilización de un sistema de estadísticas que nos ayuden a monitorizar nuestras aplicaciones ya que nos permitirá saber cual es la utilización que se esta haciendo de nuestra aplicación y distinguir diferentes perfiles de usuario.

Después de analizar las alternativas existentes en el mercado, se ha decido implementar Flurry. El proceso de implementación de Flurry es extremadamente sencillo.

Una vez descargado el SDK, tan solo tenemos que copiar a nuestro proyecto estos tres ficheros:

ProjectApiKey.txt: Fichero que contiene el nombre del proyecto y el API Key que nos proporcionan al registrarnos en la página web.

FlurryAnalytics.h: Fichero de cabeceras que contiene los métodos para el funcionamiento de Flurry Analytics.

libFlurryAnalytics.a: Librería que contiene las colecciones necesarias y el código de comunicación con los servidores.

Después de importar estos ficheros tendremos que iniciar la sesión en el delegado de la aplicación. A partir de ese momento, ya dispondremos de un sistema de estadísticas en nuestra aplicación.

Si queremos ampliar sus funcionalidades existen muchos métodos a implementar, todos ellos documentados en el SDK de Flurry.

Arquitectura

Para el correcto funcionamiento de la aplicación necesitamos de una serie de elementos que forman la arquitectura de la solución.

En la figura se muestran los dispositivos necesarios y el flujo de datos que genera la aplicación.

Podemos clasificar tres flujos de datos diferenciados en función de la actividad de la aplicación.

• Consulta de información por parte del dispositivo móvil

El terminal móvil solicitará la información al Servidor web, ya sea el parsing de un XML o la petición de recursos HTML. El Servidor web responderá a esa solicitud enviando los datos necesarios.

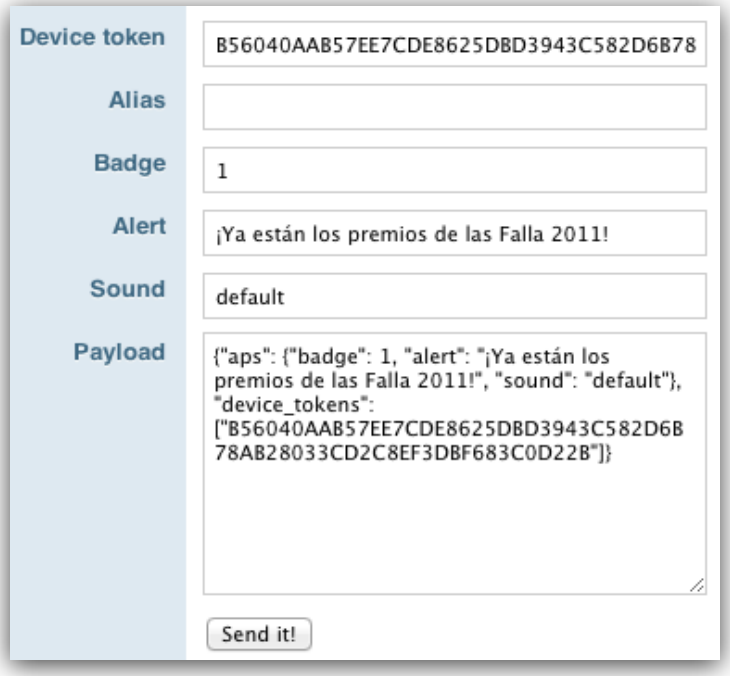
Por ejemplo, en el momento de la carga de la aplicación se realizan una petición de dos ficheros XML al Servidor web (**feed.xml** y **xmler.xml**) que contienen la información de las fallas y de los servicios, respectivamente. El Servidor web contestará a la solicitud enviando ambos ficheros. La aplicación realizará el parsing de los ficheros XML obtenidos y mostrará cuando se requiera, su contenido formateado en la aplicación.

Otra posibilidad es la consulta de ficheros HTML cuando la aplicación carga la vista de Detalle dentro de la pestaña Mapa. En ese momento, la aplicación realizará una petición de los ficheros HTML necesarios, y a medida que la información sea proporcionada por el Servidor web, los fragmentos HTML serán mostrados en la vista Detalle.

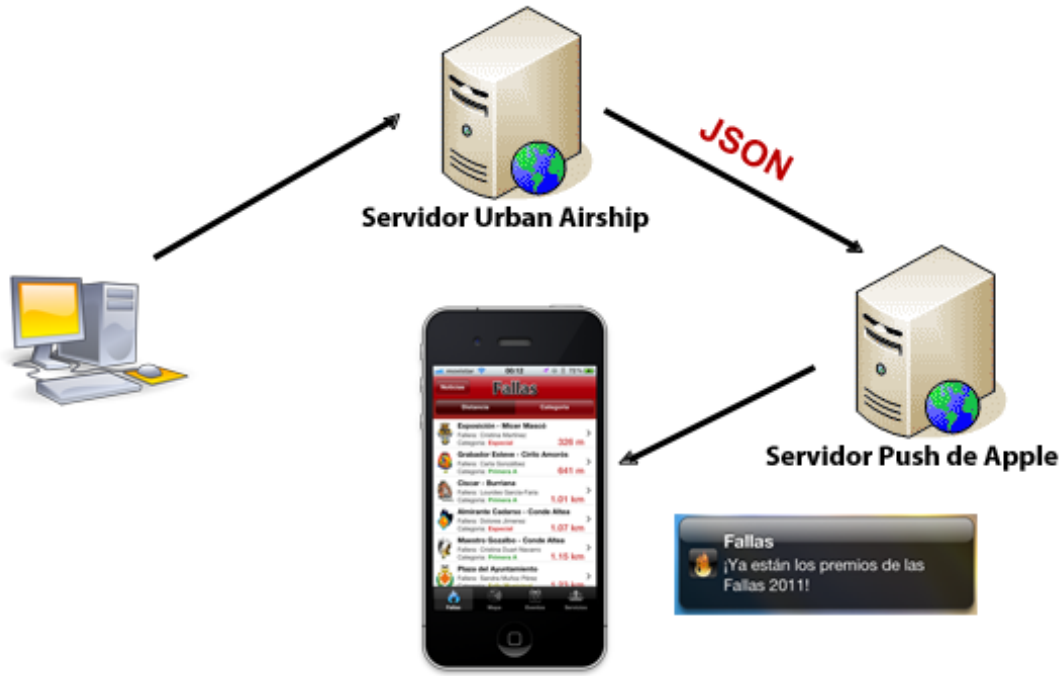
También existe la posibilidad de conectar con el Servidor web desde un dispositivo, mediante autenticación, con el fin de actualizar la información almacenada en los ficheros que alberga el servidor.

• Notificaciones Push

Para el envío de notificaciones Push debemos autenticarnos en el portal web de **Urban Airship**, este portal proporciona un servicio de notificaciones Push, previo registro de la aplicación. El portal dispone de un editor de notificaciones y un listado con los dispositivos registrados en nuestra aplicación.



Una vez redactada la notificación, el portal de Urban Airship envía la información en formato JSON al Servidor Push de Apple, y será éste quien envie la notificación final al dispositivo en cuestión.



- **Recopilación de estadísticas**

El terminal móvil enviará información estadística al Servidor Flurry, que es el servicio de estadísticas que hemos implementado en la aplicación. Su finalidad es la de realizar un análisis sobre los todos datos proporcionados por los dispositivos que hagan uso de la aplicación. De forma que podremos consultar estas estadísticas a través de la página **dev.flurry.com** mediante un dispositivo autenticado.

Diseño e Implementación de la Aplicación

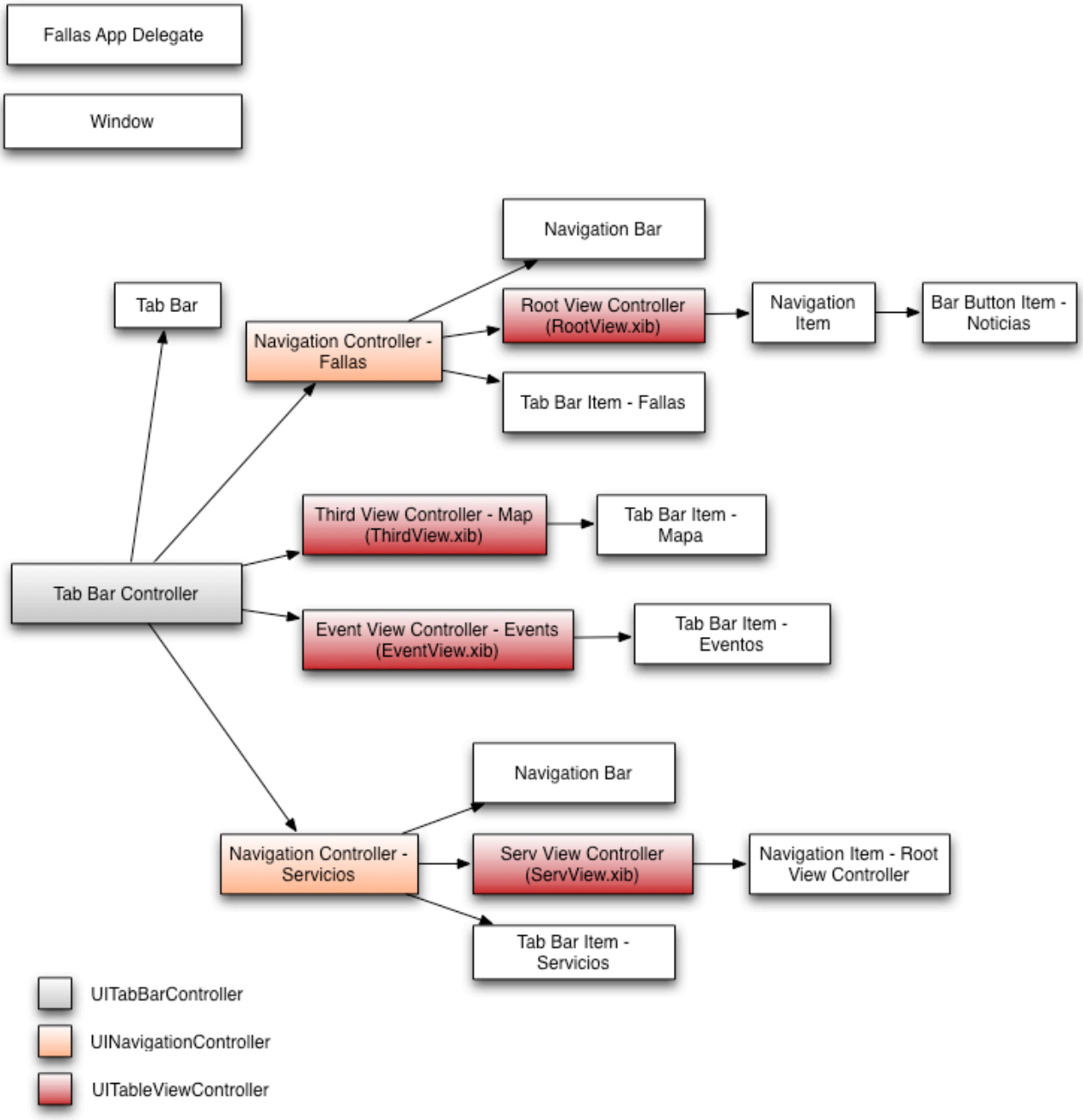
Al crear un proyecto para la creación de una aplicación iOS se generan automáticamente una serie de ficheros. De ellos, hay dos que se encargan de iniciar los procesos necesarios para el funcionamiento de la aplicación.

Por un lado, los ficheros **AppDelegate.h** y **AppDelegate.m**, que gestionan el ciclo de vida de la aplicación y que posee los métodos que se ejecutan durante los diferentes estados de la aplicación como puede ser al inicio, al entrar en un estado de inactividad, al volver de él, etc... por otro lado, se encuentra el fichero **MainWindow.xib** que gestiona la interfaz gráfica de la aplicación.

A partir de ahí se van agregando nuevos objetos que dan funcionalidad a la aplicación y que son invocados desde el AppDelegate.

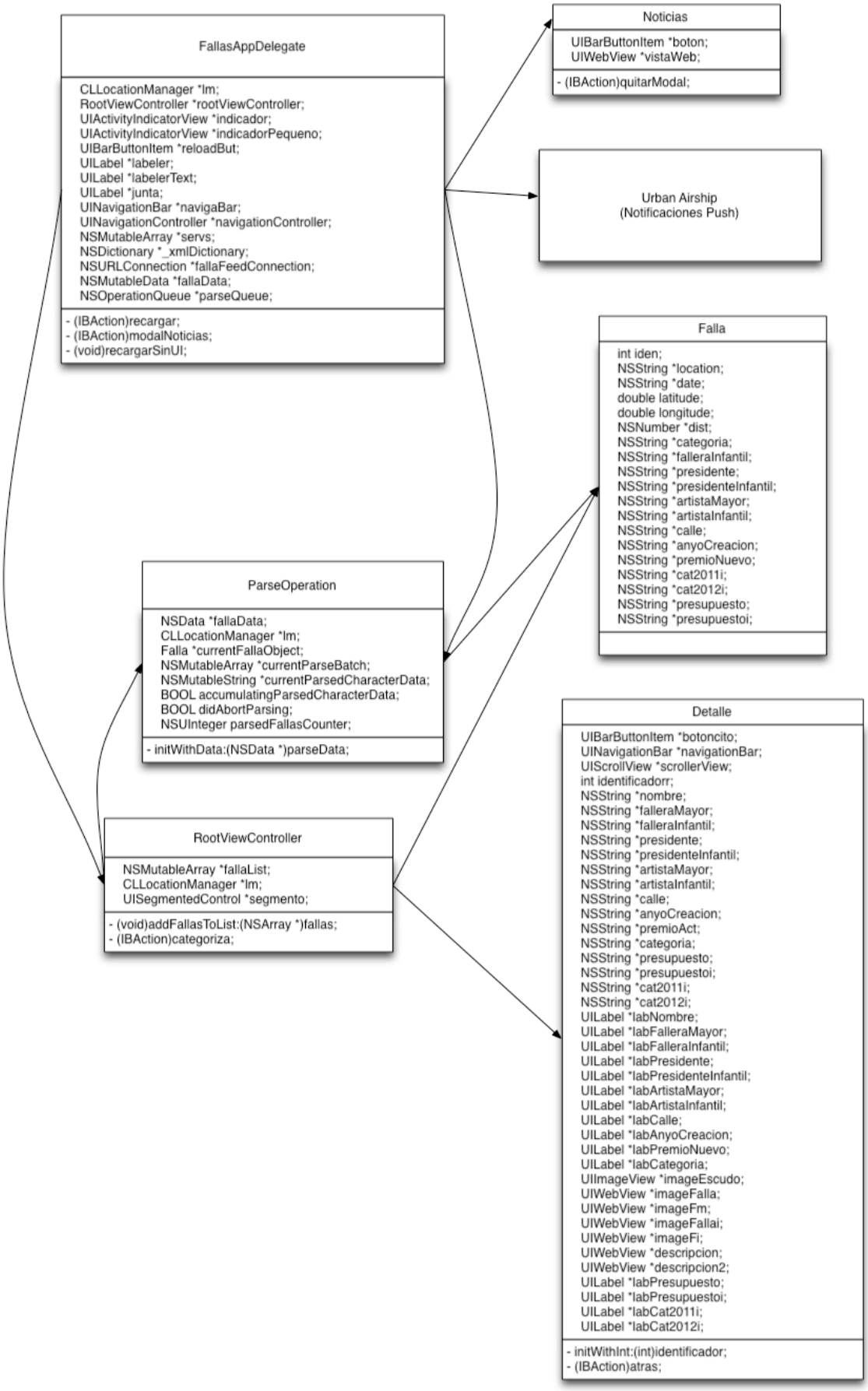
MainWindow.xib

El fichero MainWindow.xib contiene los elementos gráficos que se mostrarán en la aplicación y tiene la siguiente estructura:



La estructura que se muestra es la jerarquía de elementos gráficos que se generan al inicio de la aplicación. La información que contienen estos elementos es estática, menos en los controladores de vistas que delegan en otras vistas (fondo rojo), estas vistas serán generadas por sus propios controladores, convirtiendo a estas vistas en dinámicas.

1. Pestaña Fallas



- **FallasAppDelegate.h/FallasAppDelegate.m**

El delegado de la aplicación es siempre un objeto genérico, es decir que hereda de **NSObject**, e implementa el protocolo **UIApplicationDelegate**.

En nuestro caso vamos a implementar algunos protocolos más: **UITabBarControllerDelegate**, **CLLocationManagerDelegate** y **UIAccelerometerDelegate**.

- **UITabBarControllerDelegate**: Protocolo que se encarga de la gestión de la navegación mediante barra de pestañas, que en nuestro caso es el tipo de navegación que vamos a emplear.

- **CLLocationManagerDelegate**: Protocolo que se encarga de la gestión de los eventos de localización. A partir de la implementación de este protocolo podemos crear una instancia de un objeto de tipo **CLLocationManager**, el cual dispone de los métodos necesarios para calcular la posición actual del dispositivo y todas las operaciones relativas.

- **UIAccelerometerDelegate**: Protocolo que se encarga de la gestión de los sensores de movimiento de los que dispone el terminal, el cual define los métodos necesarios para la obtención de los datos que proporcionan los sensores en los tres ejes.

Además de la implementación de estos tres protocolos, se requiere de la importación de ciertos frameworks para capacitar al delegado de los recursos necesarios.

Los frameworks importados en este caso son:

- **UIKit**: Es el framework que se encarga de la interfaz gráfica.
- **CoreLocation**: Es el framework que se encarga del sistema de localización.
- **QuartzCore**: Es el framework que permite la utilización de gráficos avanzados.
- **CFNetwork**: Es el framework que permite la monitorización del tráfico de red.

Y también los ficheros .h de las clases que vayamos a utilizar, que son las siguientes:

- **UIAirship.h** y **UAPush.h**: Estas clases implementan las notificaciones Push.

- **Noticias.h**: Clase que implementa la vista Noticias.

- **Falla.h**: Clase que implementa el objeto Falla que contiene todos los atributos de cada una de las fallas.

- **RootViewController.h**: Clase que implementa la vista Fallas que es el listado principal que se muestra en la primera pestaña.

- **ParseOperation.h**: Clase de tipo NSOperation que realiza las operaciones necesarias para el parsing del fichero feed.xml y almacenar su contenido en un objeto de tipo Falla.

- **ServViewController.h**: Clase que implementa la vista Servicios que es un listado con los servicios que se muestran en la cuarta pestaña.

- **XMLReader.h**: Clase que almacena en un diccionario (NSDictionary) la información analizada en el fichero xmler.xml y que contiene la información de los servicios que se mostrarán en la pestaña Servicios.

Detalles de implementación:

didFinishLaunchingWithOptions

Es el método que se ejecuta al iniciar la aplicación y por consiguiente debe gestionar los procesos necesarios para el funcionamiento de la aplicación.

- Crea e inicializa el objeto que permite la localización, con un filtro de 50 metros. El propósito de este filtro es minimizar el consumo de energía y procesamiento, ya que solo se llamará al método que calcula los datos visibles en la aplicación, cuando las medidas obtenidas por el gps difieran en más de 50 metros.

- Creamos una pequeña animación que se mostrará en pantalla durante el procesamiento de los datos.

- Iniciamos el proceso de parsing del fichero **feed.xml** que contiene los datos de las fallas a procesar.

- Iniciamos el proceso de parsing del fichero **xmler.xml** que contiene los datos de los servicios que se mostrarán en la pestaña **Servicios**.

- Iniciamos los procesos necesarios para que **Urban Airship**, que es el sistema de notificaciones, se conecte a los servidores apropiados.

- Superponemos una imagen sobre el título de la barra de navegación, ya que solo es posible utilizar las fuentes que Xcode permite, en cambio si superponemos una imagen, podemos crear el logo que queramos.

- Se registra si la entrada a la aplicación se ha hecho mediante una notificación Push, en caso de que haya sido así, se presentará sobre la vista principal, la vista de noticias.

Además del método de inicio, hemos realizado algunas implementaciones en otros métodos:

didRecieveRemoteNotification

Este método se ejecutará cuando la aplicación recibe una notificación, pero se encuentra en segundo plano. El cometido de este método es mostrar la vista de Noticias. Básicamente es lo mismo que habíamos hecho en **didFinishLaunchingWithOptions**, la diferencia reside en que uno se ejecutará únicamente al inicio de la aplicación, mientras que el otro se ejecutará una vez la aplicación ya este iniciada.

applicationWillEnterForeground

Este método se ejecutará cuando la aplicación vaya a entrar en modo primer plano. Lo que debemos hacer es asegurarnos de desactivar todas las animaciones de carga y detener el proceso de localización. Esto es necesario, porque si hay una transición rápida entre diferentes

estados es posible que creemos dos instancias de la animación, y cuando acabase de cargar, sólo se eliminaría una de ellas, quedando una animación visible permanentemente. Por lo que de esta forma nos aseguramos que antes de volver a la aplicación no haya animaciones en ejecución.

applicationDidBecomeActive

Este método se ejecuta cuando la aplicación pasa a ser la aplicación en primer plano. Y lo que haremos es eliminar el badge, que es el número de notificaciones que tenemos pendientes de leer que aparecen en el icono de la aplicación. Y también ejecutaremos el método recargar, que básicamente tiene las mismas instrucciones que **didFinishLaunchingWithOptions** pero con unas animaciones menos intrusivas.

tabBarController didSelectViewController

Es el método delegado opcional de **UITabBarControllerDelegate**, y nos permite identificar en que pestaña estamos actualmente de forma en que si nos encontramos en la pestaña 2, que es la pestaña de Mapa, tendremos en cuenta el valor del acelerómetro (UIAccelerometer), de forma en que si ponemos el terminal en posición vertical, sólo será tenido en cuenta por el controlador de vista de la pestaña 2, que en este caso es **ThirdViewController**.
También eliminaremos unos tweaks gráficos que sólo queremos que estén visibles en la primera pestaña. El tweak consiste en un gráfico que nos permite eliminar la separación entre la barra de navegación y la vista **RootViewController**, de esta forma conseguimos una interfaz innovadora y nunca vista hasta ahora.

Existen otros métodos que se encargan básicamente de gestionar los eventos que se producen en caso de que haya un fallo en la conexión.

• Fallas.h/Fallas.m

Es una clase que hereda de **NSObject** y que no presenta métodos.
Su propósito es almacenar la información de cada una de las fallas. Existen 19 atributos distintos en el objeto Falla, como pueden ser el identificador, latitud, longitud, nombre, categoría, etc...

• Noticias.h/Noticias.m

Es una clase de tipo **UIViewController**, es decir, una vista normal y corriente, que carga una página web específica, la cual contendrá la última noticia redactada. Es por ello que se precisa de las notificaciones Push para notificar al usuario de que la noticia ha sido actualizada.

• ParseOperation.h/ParseOperation.m

Es una clase de tipo **NSOperation** y cuyo cometido es analizar el fichero feed.xml que contiene los datos de las fallas y almacenar la información en un objeto de tipo Falla, a continuación almacena todos los objetos de tipo Falla en un array (NSArray) al cual tendrá

acceso la vista que muestra el listado con las fallas, en este caso **RootViewController**. El proceso de parsing es el siguiente, definimos de forma estática patrones de texto a identificar. A continuación, se implementan los métodos necesarios para cumplimentar el protocolo **NSXMLParserDelegate**. A través de estos métodos vamos almacenando la información en la variable correspondiente del objeto de tipo Falla. La llamada a esta operación se lleva a cabo en **FallasAppDelegate**.

El fichero que analiza la clase **NSOperation** es **feed.xml** que está hospedado en un servidor web y tiene la siguiente estructura:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:georss="http://
www.georss.org/georss">
<entry>
      <id>3</id>
      <title>Convento Jerusalen - M. Marzal</title>
      <falleraMayor>María Gonzalez Villena</falleraMayor>
      <falleraInfantil>Ana Pons Tomás</falleraInfantil>
      <presidente>Vicente Gil Pérez</presidente>
      <presidenteInfantil>Roberto Gómez Colomer</presidenteInfantil>
      <artistaMayor>Pedro Santaeulalia</artistaMayor>
      <artistaInfantil>Paloma Roig</artistaInfantil>
      <anoCreacion>1925</anoCreacion>
      <descripcion></descripcion>
      <descripcion2></descripcion2>
      <calle>Convento Jerusalen - Matemático Marzal</calle>
      <cat2011>1º de Especial</cat2011>
      <cat2011i></cat2011i>
      <presupuesto></presupuesto>
      <presupuestoi></presupuestoi>
      <cat2012i></cat2012i>
      <cat2012>Especial</cat2012>
      <georss:point>39.466456 -0.379511</georss:point>
</entry>
</feed>
```

• RootViewController.h/RootViewController.m

Es una clase de tipo **UITableViewController**, es decir una vista de tabla, y su función es mostrar en cada una de las celdas de la tabla la información correspondiente a cada falla.

La plantilla de las clases **UITableViewController** incluyen métodos a implementar que indican el número de filas por sección, el contenido de una celda dado un índice o la acción a realizar una vez elegida una celda.

El proceso en este caso consiste en extraer la información de la falla que ocupa la posición i en el array, siendo i el índice introducido como parámetro. Una vez extraída la información, se le da formato con diferentes tamaños y colores.

Como tenemos acceso a la posición actual del dispositivo y también tenemos las coordenadas de cada falla, podemos llamar al método que calcula la distancia entre ambos. Una vez calculadas todas las distancias, podemos reordenar el array por distancia usando un objeto de tipo **NSSortDescriptor**. He añadido un selector que permite elegir dos criterios de ordenación. El primero es ordenar por distancia, que es el elegido por defecto y otro que ordena por categoría, en este caso he utilizado el identificador como criterio de ordenación, pero bien podría haber sido elegido otro atributo del objeto Falla como por ejemplo, por presupuesto.

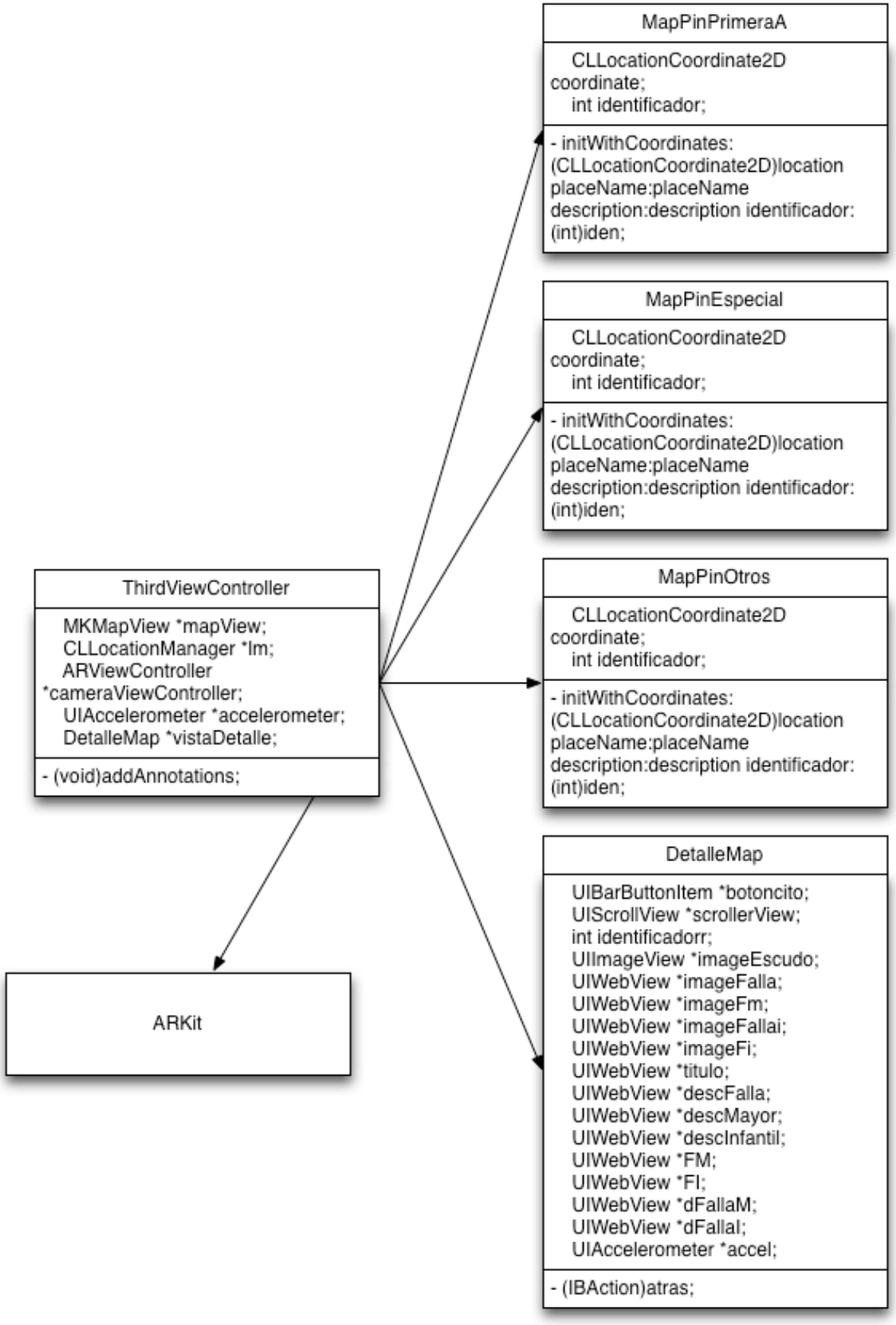
Si se pulsa sobre alguna de las celdas, aparecerá una nueva vista con dos opciones, Información, que mostrará una nueva vista modal de tipo **Detalle** o **Como llegar**, que no es más que un link a Google Maps, donde se indica en la url las coordenadas de origen (la posición del dispositivo) y las coordenadas de la falla destino. El dispositivo interpreta los links a Google Maps de forma distinta al resto de links y en lugar de abrir un navegador web, ejecuta la aplicación nativa Mapas.

• Detalle.h/Detalle.m

Es una clase de tipo **UIViewController**, que muestra una ficha detallada de la falla seleccionada. Todos los atributos que se muestran en la ficha han sido pasados a través del objeto Falla, a excepción de las fotos, que se cargarán como una vista web, de este modo evitamos tener que almacenar todas las fotos de forma loca, consiguiendo una aplicación más liviana.

Otro problema asociado a la carga de imágenes de forma local es que la vista no será mostrada hasta que todos los elementos de la interfaz estén cargados. En cambio si la carga se realiza a través de una vista web, podemos cargar el resto de componentes de forma asíncrona, por lo que el texto aparece de forma instantánea y las imágenes se van cargando como ocurre en un navegador. El uso de la red, que generalmente tiene una prestaciones inferiores al almacenamiento local, en este caso resulta una ventaja.

2. Pestaña Mapa



• ThirdViewController.h/ThirdViewController.m

Es una vista de tipo **UIViewController** que implementa los protocolos delegados **CLLocationManagerDelegate**, **MKMapViewDelegate**, **UIAccelerometerDelegate**, **ARLocationDelegate**.

El primero de ellos, como se he visto anteriormente, gestiona los procesos de localización. **MKMapViewDelegate** se encarga de la gestión de los procesos asociados a la vista de mapa (MKMapView). **UIAccelerometerDelegate** se encarga de la gestión del acelerómetro y **ARLocationDelegate** es el protocolo que permite ejecutar los procesos de Realidad Aumentada.

Básicamente lo que ofrece esta vista es un mapa proporcionado por **MKMapView** donde se muestran las fallas mediante pines. Para definir estos pines necesitamos implementar algunos de los métodos de **MKMapViewDelegate**, donde tenemos que crear un objeto de tipo **MKAnnotation** el cual incluye información como el nombre o sus coordenadas.

En caso de querer personalizar estos pines, deberemos de crear un objeto heredado de **MKAnnotation**. En nuestro caso, como queremos representar las fallas de Especial con pines de color rojo, las fallas de Primera A con pines de color verde y la falla Municipal con un pin morado, deberemos de crear tres nuevas clases que hereden de **MKAnnotation, MapPinEspecial**, **MapPinPrimeraA** y **MapPinOtros**.

Para indicar con claridad que color representa cada tipo de falla, hemos creado unas etiquetas a modo de leyenda en la parte superior, estas etiquetas se encuentran superpuestas sobre la vista de mapa.

La utilización del protocolo **MKMapViewDelegate** permite la localización del dispositivo sin tener que recurrir a **CLLocationManagerDelegate**, pero en esta ocasión hemos implementado ambos ya que si utilizamos únicamente **MKMapViewDelegate** no tenemos la posibilidad de centrar el mapa con la posición en la que nos encontramos, ya que no proporciona coordenadas, simplemente una referencia gráfica independiente del resto del procesamiento. En cambio se dispone de las coordenadas del dispositivo mediante **CLLocationManagerDelegate** si que es posible llamar a la función necesaria para centrar el mapa dadas una posición concreta.

El proceso de añadir pines con su correspondiente información al mapa se puede realizar de diferentes formas. Una posibilidad es: ya que tenemos la información de cada falla, incluida sus coordenadas en un fichero xml, podemos realizar el parsing de los datos y asignarlos a un objeto de tipo **MKAnnotation**. En cambio, como la carga del mapa es el proceso que más recursos consume de la aplicación, debemos minimizar, en la medida que podamos, el resto de cálculos para poder ofrecer la mejor experiencia de usuario posible. Además, las coordenadas no van a ser cambiantes, por lo que no es necesario consultar la información cada vez que se ejecute la aplicación, en cambio, si almacenamos las coordenadas de forma estática conseguimos una velocidad de procesamiento mayor y en caso de que hubiese que actualizar alguna coordenada, sería necesario una actualización de la aplicación. Pensamos que es el precio a pagar por ofrecer la máxima fluidez, y compensa con creces ya que la frecuencia de actualización de las coordenadas es muy baja.

Al tener que implementar el protocolo **UIAccelerometerDelegate** debemos completar el método **accelerometer didAccelerate**. La implementación del método consistirá en que si se registra un valor del eje z del acelerómetro mayor que -0.1 entonces se creará una instancia de tipo **ARViewController**, es decir, entraremos en el modo de realidad aumentada.

- **MapPinEspecial.h/MapPinEspecial.m**
- **MapPinPrimeraA.h/MapPinPrimeraA.m**
- **MapPinOtros.h/MapPinOtros.m**

MapPin* es en realidad un objeto que hereda de NSObject y que implementa el protocolo MKAnnotation.

Básicamente, su implementación se reduce a definir un método de inicialización en el que se pasan como parámetros las coordenadas, el nombre, la descripción, y en nuestro caso, un identificador.

Al tener diferentes clases de pines, podemos identificar dentro de ThirdViewController que tipo de objeto es cada pin y asignar un color distinto en función de ello.

- **DetalleMap.h/DetalleMap.m**

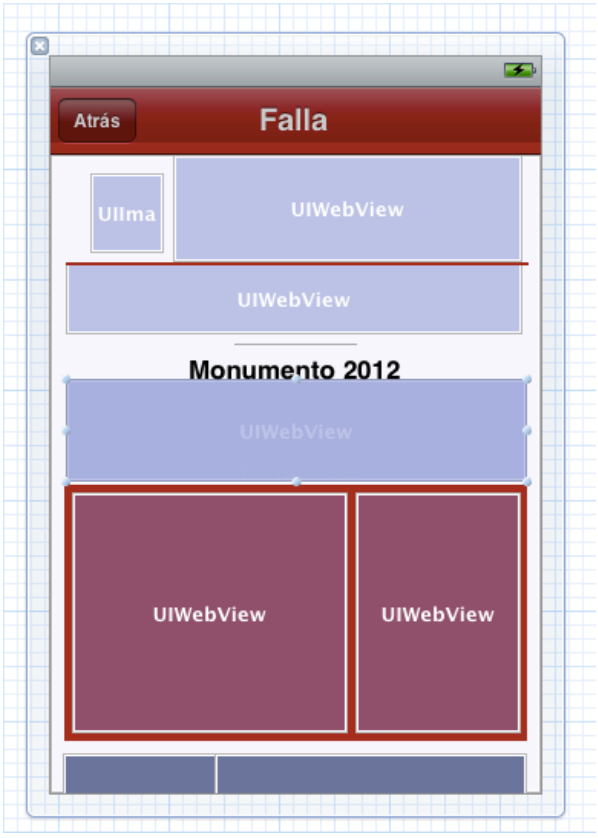
DetalleMap es una clase de tipo UIViewController, es decir, una vista plana, sobra la cual vamos a añadir todos los elementos necesarios para crear una ficha de falla. El aspecto de esta vista es exactamente igual al creado con la clase Detalle, pero en este caso, no podemos pasarle a la vista todos los parámetros que necesita la vista Detalle, porque el método que se ejecuta al pulsar sobre el indicador que genera el pin no permite parámetros. Por ello nos vemos forzados a crear una vista idéntica a Detalle y que tenga como identificador del objeto el identificador de la falla. De esta forma, si extraemos los cálculos del código del controlador de vista y los portamos a un servidor web, nos bastará únicamente con el identificador.

Por tanto en lugar de obtener los valores de la falla y representarlos sobre etiquetas, en este caso vamos a cargar pequeñas vistas web donde se mostrará la información correspondiente. Así que el procesamiento de la información se realiza en el servidor web. Cada uno de estros fragmentos web dispone del código necesario para analizar la información del XML y convertirla en cadenas de texto.

Tan solo tendremos que cargar las páginas web pasando como argumento el identificador que corresponda:

```
paginaWeb = [[NSURL alloc] initWithString:[NSString
stringWithFormat:@"http://servidorweb.com/Fallas/feed/descFalla.php?id=
%d",identificador]];
[descFalla loadRequest:[NSURLRequest requestWithURL:paginaWeb]];
[[[descFalla subviews] lastObject] setScrollEnabled:NO];
[paginaWeb release];
```

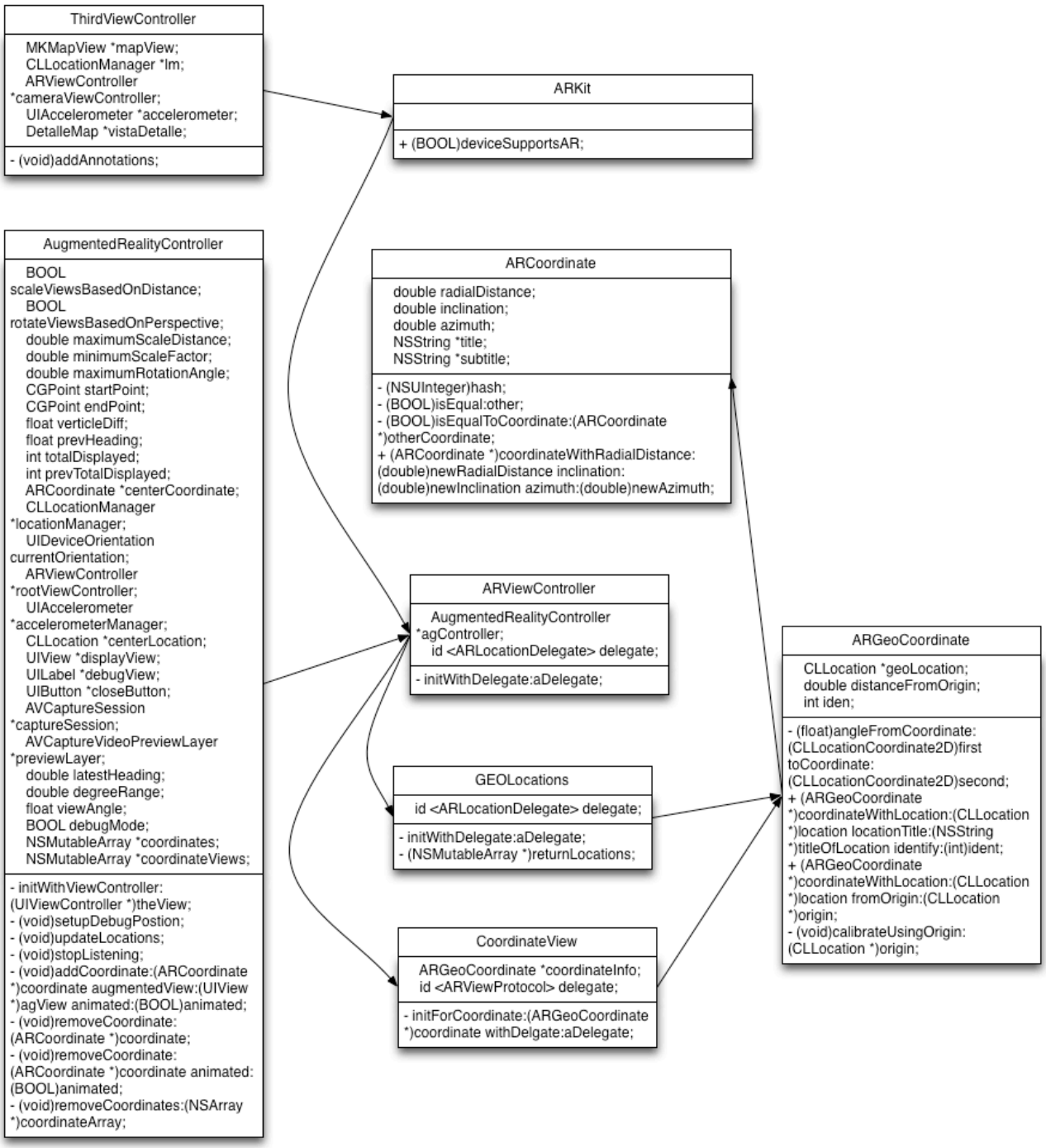
Detalle.xib **DetalleMap.xib**



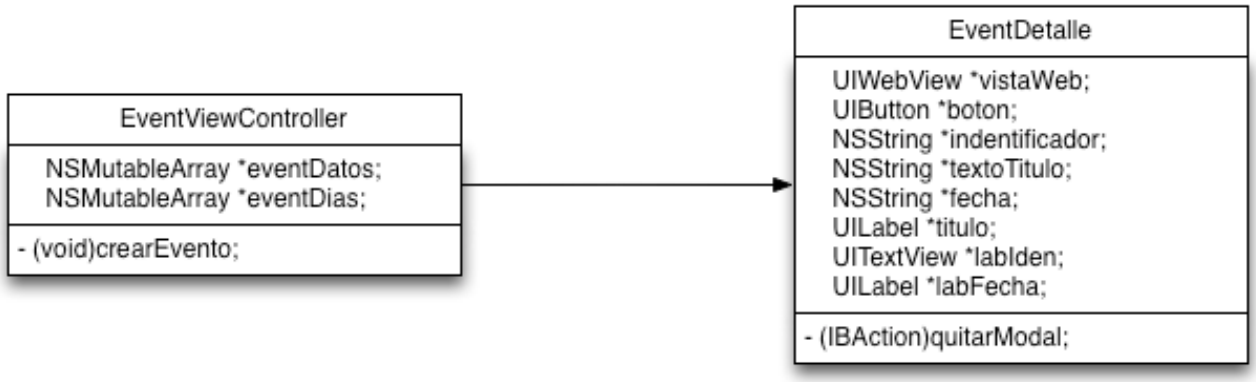
Código php que contiene la información Premio 2012, Premio 2011 y Presupuesto:

```
<?php
	$url = "feed.xml";
	$contenido_xml = "";
	if($d = fopen($url, "r")) {
		while ($aux=fgets($d,2048)) {
			$contenido_xml .= $aux;
		}
		fclose($d);
	}
	else {
		echo "No se pudo abrir el XML";
	}
	$xml = simplexml_load_string($contenido_xml);
	?>
	<?php $var1 = $_GET['id']; ?>
	<font size="3"><b>Premio 2012:</b>
<?php echo utf8_decode($xml->entry[$var1-1]->cat2012); ?><br />
	<b>Premio 2011:</b> <?php echo utf8_decode($xml->entry[$var1-1]-
>cat2011); ?><br />
	<b>Presupuesto:</b> <?php echo utf8_decode($xml->entry[$var1-1]-
>presupuesto); ?></font>
```

ARKit


ThirdViewController
MKMapView *mapView;
CLLocationManager *lm;
ARViewController *cameraViewController;
UIAccelerometer *accelerometer;
DetalleMap *vistaDetalle;
- (void)addAnnotations;
ARKit
+ (BOOL)deviceSupportsAR;
AugmentedRealityController
BOOL scaleViewsBasedOnDistance;
BOOL rotateViewsBasedOnPerspective;
double maximumScaleDistance;
double minimumScaleFactor;
double maximumRotationAngle;
CGPoint startPoint;
CGPoint endPoint;
float verticleDiff;
float prevHeading;
int totalDisplayed;
int prevTotalDisplayed;
ARCoordinate *centerCoordinate;
CLLocationManager *locationManager;
UIDeviceOrientation currentOrientation;
ARViewController *rootViewController;
UIAccelerometer *accelerometerManager;
CLLocation *centerLocation;
UIView *displayView;
UILabel *debugView;
UIButton *closeButton;
AVCaptureSession *captureSession;
AVCaptureVideoPreviewLayer *previewLayer;
double latestHeading;
double degreeRange;
float viewAngle;
BOOL debugMode;
NSMutableArray *coordinates;
NSMutableArray *coordinateViews;
- initWithViewController:(UIViewController *)theView;
- (void)setupDebugPostion;
- (void)updateLocations;
- (void)stopListening;
- (void)addCoordinate:(ARCoordinate *)coordinate augmentedView:(UIView *)agView animated:(BOOL)animated;
- (void)removeCoordinate:(ARCoordinate *)coordinate;
- (void)removeCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated;
- (void)removeCoordinates:(NSArray *)coordinateArray;
ARCoordinate
double radialDistance;
double inclination;
double azimuth;
NSString *title;
NSString *subtitle;
- (NSUInteger)hash;
- (BOOL)isEqual:other;
- (BOOL)isEqualToCoordinate:(ARCoordinate *)otherCoordinate;
+ (ARCoordinate *)coordinateWithRadialDistance:(double)newRadialDistance inclination:(double)newInclination azimuth:(double)newAzimuth;
ARViewController
AugmentedRealityController *agController;
id <ARLocationDelegate> delegate;
- initWithDelegate:aDelegate;
GEOLocations
id <ARLocationDelegate> delegate;
- initWithDelegate:aDelegate;
- (NSMutableArray *)returnLocations;
CoordinateView
ARGeoCoordinate *coordinateInfo;
id <ARViewProtocol> delegate;
- initForCoordinate:(ARGeoCoordinate *)coordinate withDelgate:aDelegate;
ARGeoCoordinate
CLLocation *geoLocation;
double distanceFromOrigin;
int iden;
- (float)angleFromCoordinate:(CLLocationCoordinate2D)first toCoordinate:(CLLocationCoordinate2D)second;
+ (ARGeoCoordinate *)coordinateWithLocation:(CLLocation *)location locationTitle:(NSString *)titleOfLocation identify:(int)ident;
+ (ARGeoCoordinate *)coordinateWithLocation:(CLLocation *)location fromOrigin:(CLLocation *)origin;
- (void)calibrateUsingOrigin:(CLLocation *)origin;

3. Pestaña Eventos



- **EventViewController.h/EventViewController.m**

Es una clase de tipo **UITableViewController** que muestra en un listado con los eventos agrupados por días. Los eventos no son más que diccionarios (NSMutableDictionary) con las claves título y identificador, creados en tiempo de ejecución. Esos diccionarios se almacenan en arrays (NSArray) que representan cada uno de los días.

Por tanto, el diccionario únicamente contiene el título del evento y su identificador, no alberga información alguna más que el título. En el momento en el que el usuario seleccione una de las celdas de la tabla, se creara una instancia de tipo **EventDetalle**.

- **EventDetalle.h/EventDetalle.m**

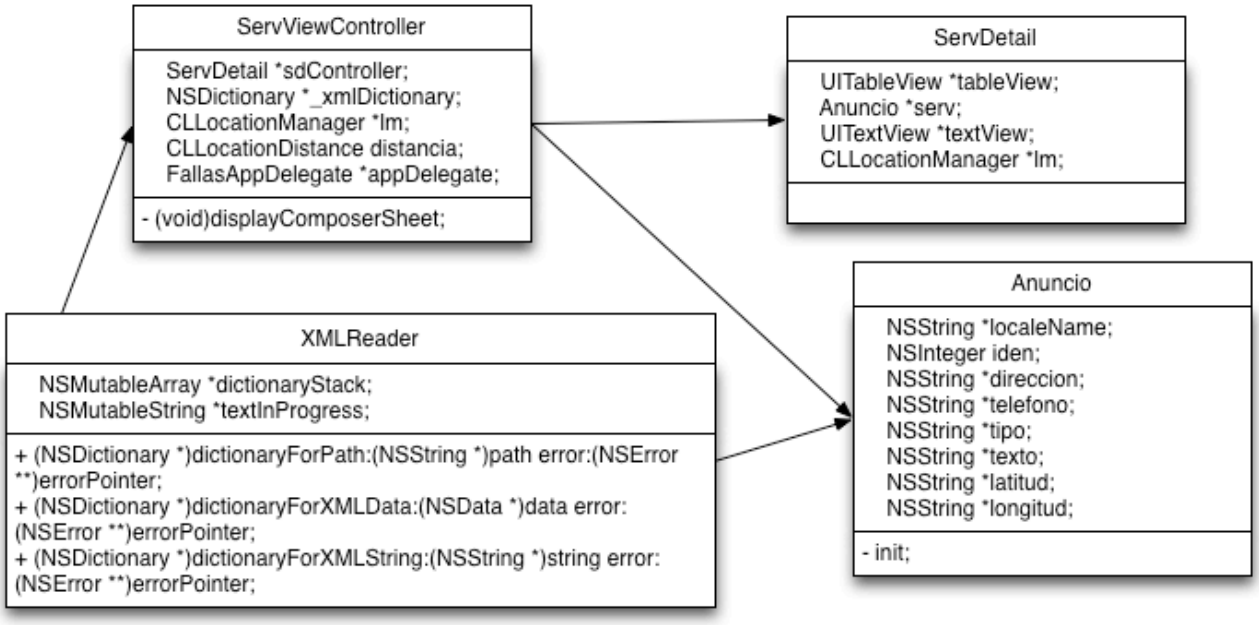
Es una clase de tipo **UIViewController**, que mostrará el título del evento y una vista web con la información del evento. Estas vistas web con puro html, lo que significa que tendremos que crear para cada evento un fichero html con su información.

```
NSURL *paginaWeb = [[NSURL alloc] initWithString:[NSString
stringWithFormat:@"http://servidorweb.com/Fallas/eventos/evento
%@.html",identificador]];
    [vistaWeb loadRequest:[NSURLRequest requestWithURL:paginaWeb]];
    [paginaWeb release];
```

En el caso de disponer de un evento con identificador 3, por ejemplo, deberemos de crear un fichero evento3.html que contenga la información necesaria a mostrar.

De esta forma, la creación de eventos se realiza a través del código de la aplicación, por tanto, si queremos agregar nuevos eventos, deberemos actualizar la aplicación. Sin embargo, el contenido del evento, al estar implementado en html, puede ser actualizado en cualquier momento y de forma transparente para el usuario.

4. Pestaña Servicios



- **ServViewController.h/ServViewController.m**

Es una clase de tipo **UITableViewController** que implementa los protocolos **CLLocationMangerDelegate** y **MFMailComposeViewControllerDelegate**.

Si misión consiste en mostrar un listado con los eventos proporcionados por el ficher xmler.xml. Esta información será suministrada por XMLReader en forma de diccionario (NSDictionary).

Para dar formato a las celdas de la tabla, ServViewController convierte el diccionario en un array (NSArray) de forma en que podemos clasificar los servicios en función de su tipo. También calcularemos la distancia a la que se encuentra el servicio y la mostraremos en su correspondiente celda.

En caso de que el usuario pulse sobre una de las celdas, se creará una instancia de tipo Anuncio que contiene toda la información del servicio y otra de tipo ServDetail a la que pasaremos el Anuncio recién creado.

Se ha creado un botón en la barra de navegación con la etiqueta **Anúnciate**, que hace uso del protocolo **MFMailComposeViewControllerDelegate** y permite el envío de correos electrónicos desde la propia aplicación con un asunto y contenido predeterminado. En este caso, para anunciar la posibilidad de estar interesado en aparecer en el listado de servicios.

- **XMLReader.h/XMLReader.m**

Es un fichero de tipo **NSObject** que implementa el protocolo **NSXMLParserDelegate**, el cual tiene la función de realizar el parsing de ficheros XML.

Esta clase no forma parte de ninguna plantilla de Xcode, sino que es una herramienta opensource disponible en internet ampliamente utilizada por la comunidad y cuya finalidad es almacenar el contenido de un fichero XML en un diccionario, de tal forma que no es necesaria la modificación del fichero ya que es independiente del XML.

Al inicio de la aplicación, **FallasAppDelegate** crea una instancia de XMLReader sobre el fichero **xmler.xml** de forma en que el proceso de parsing se realiza durante el inicio de la ejecución de la aplicación. Así cuando el usuario accede a la pestaña Servicios, la aplicación ya ha procesado la información necesaria, de modo que no se producen cálculos durante la navegación, mejorando así la experiencia de usuario.

El fichero **xmler.xml** tiene la siguiente estructura:

```
<?xml version="1.0" encoding="UTF-8"?>
<Anuncios>
	<Restaurantes id="3">
		<localeName>Kentucky Fried Chicken</localeName>
		<direccion>Gran Vía del Marqués del Turia, 56</direccion>
		<latitud>39.467083</latitud>
		<longitud>-0.367758</longitud>
		<tipo>Restaurante</tipo>
		<telefono>345</telefono>
		<texto></texto>
	</Restaurantes>
	<Decoracion id="4">
		<localeName>Leroy Merlin</localeName>
		<direccion>Port Saplaya</direccion>
		<latitud>39.504902</latitud>
		<longitud>-0.323861</longitud>
		<tipo>Decoración</tipo>
		<telefono>456</telefono>
		<texto>Hola hola caracola</texto>
	</Decoracion>
</Anuncios>
```

- **Anuncio.h/Anuncio.m**

Es un fichero de tipo **NSObject** que tiene como atributos las propiedades de un anuncio mostrado en el listado de servicios.

Las propiedades de Anuncio son: el nombre, el teléfono, la dirección, el identificador, el tipo, la descripción, la longitud y la latitud.

- **ServDetail.h/ServDetail.m**

Es un fichero de tipo **UITableViewController** que consta de tres secciones y una celda por cada sección. Los títulos de las secciones son: Nombre, Teléfono y Dirección. El contenido de las celdas se obtiene del objeto Anuncio. Además se ha añadido una vista debajo de la tabla donde se muestra la descripción del anuncio.

Si se pulsa sobre las celdas se realizarán diferentes acciones, en el caso del nombre, no se registra el pulso, pero en caso de ser el teléfono, se llamará directamente al número indicado, y si se pulsa sobre la dirección, se calculará la ruta desde la posición actual hasta la dirección indicada.

```
if (indexPath.section == 1) {
        NSString *telefonor = [NSString stringWithFormat:@"tel:
%@",serv.telefono];
        [[UIApplication sharedApplication] openURL:[[NSURL alloc]
initWithString:telefonor]];
    }
    else if (indexPath.section == 2) {
        NSString *mapsQuery =
        [NSString stringWithFormat:@"http://maps.google.com/maps?saddr=%f,
%f&daddr=%f,%f",
         lm.location.coordinate.latitude, lm.location.coordinate.longitude,
latit, longit];
        [[UIApplication sharedApplication] openURL:[NSURL
URLWithString:mapsQuery]];
    }
```

5. Pruebas

Depuración con simulador y dispositivos

A la hora de publicar una aplicación hace falta asegurarse que dicha aplicación funciona correctamente y que ofrece una experiencia de usuario satisfactoria.

Para ello es necesario establecerse unos objetivos claros y estructurar la aplicación en base a estos objetivos. Una vez hecho el diseño, el siguiente paso es la implementación. Si esta implementación se realiza de la forma más eficiente posible conseguiremos los resultados esperados.

Ahora bien, para comprobar que esta implementación cumple con nuestros requisitos hace falta una fase de testeo. Para ello, disponemos de un simulador de dispositivos en el propio Xcode. Este simulador ofrece prácticamente las mismas prestaciones que un dispositivo real, salvo algunas limitaciones.

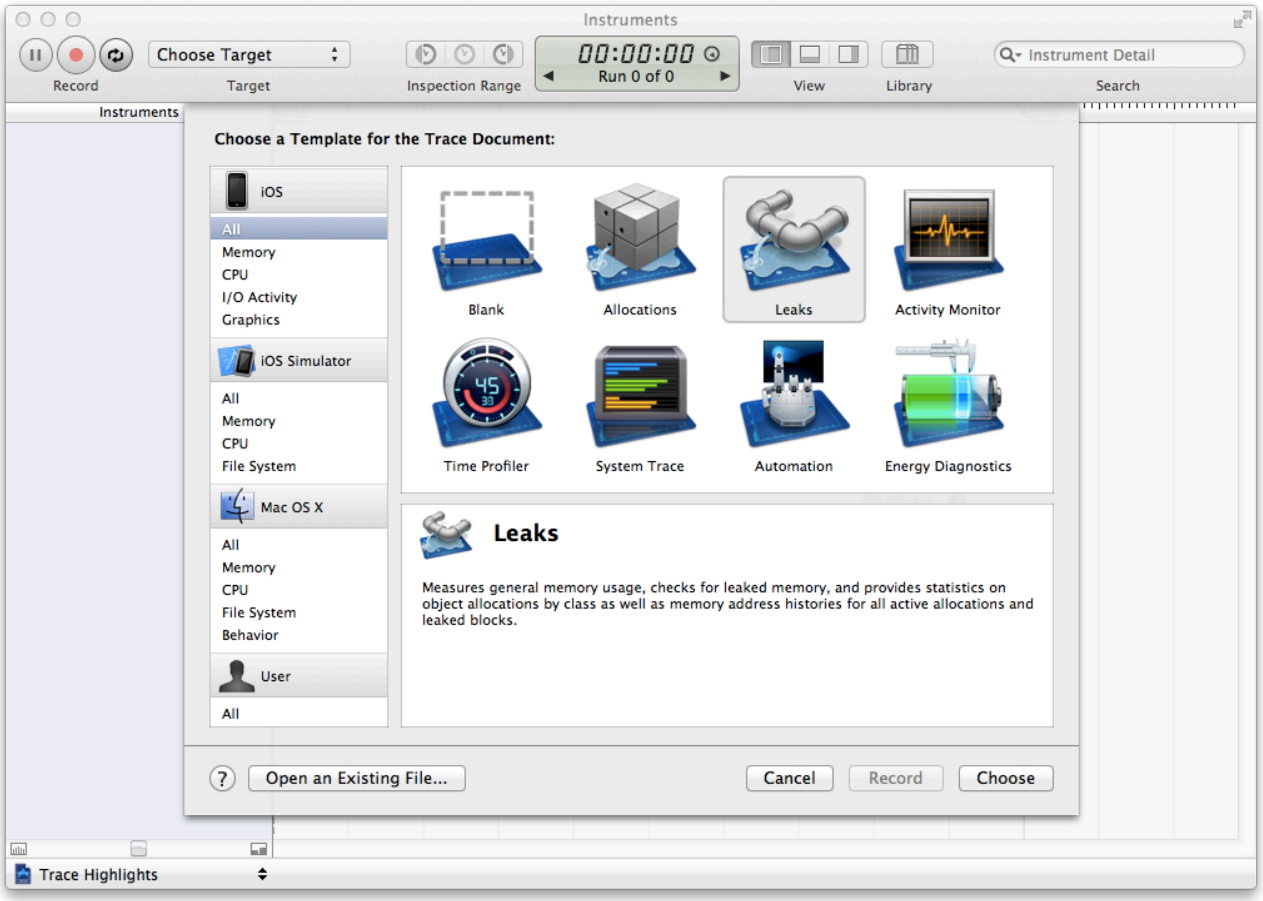
Durante la etapa de aprendizaje y el inicio de la implementación de la aplicación, el simulador ofrece todo lo necesario para el testeo de aplicaciones. A medida que la complejidad de la aplicación aumenta, existen ciertas necesidades que el simulador no puede proporcionar. A pesar de que podemos configurar la posición del simulador, no dispone de la capacidad de realizar cambios de posición *on-the-air*, por lo que no podremos comprobar si la aplicación funciona como se espera. Además, el uso del API de la cámara no está disponible en el simulador, por lo que resulta imposible comprobar el funcionamiento de la realidad aumentada.

Por ello, resulta imprescindible a la larga el uso de dispositivos reales para la fase de pruebas, ya que si la intención es la publicación de la aplicación en el App Store, necesitamos asegurarnos al 100% que la aplicación va a responder según nuestras expectativas.

Monitorizar con Instruments

La siguiente herramienta que analizaremos se llama Instruments. Instruments se utiliza para perfilar varios aspectos de la aplicación y nos ayuda a comprender el comportamiento en tiempo de ejecución de la aplicación y de iOS. Instruments forma parte de Xcode, y aunque no se encuentra integrado dentro de la aplicación, podemos encontrar el ejecutable en la carpeta /Developers/Applications.

Instruments es un flexible almacén de instrumentos de los cuales cada uno registra y muestra un aspecto diferente sobre el comportamiento de una aplicación. Basta con seleccionar los instrumentos que se quiere utilizar para capturar los aspectos particulares de la aplicación que se desea examinar.



El uso habitual que se hace de Instruments es la detección de fugas de memoria (Memory Leaks). Una fuga de memoria se da cuando la memoria es asignada por una aplicación pero nunca se libera. La forma en que Instruments sabe cuándo la aplicación tiene fugas de memoria, es mediante el control de todas las asignaciones de memoria que hace la aplicación y los punteros a dicha memoria. En el momento en que una aplicación no disponga de un puntero válido a la memoria que tenga asignada, Instruments sabe que dicha aplicación no puede liberar memoria y por consiguiente, ha tenido una fuga de memoria.

‣ Pruebas en diferentes localizaciones y dispositivos

Una vez depurada la aplicación tanto en Xcode como en un dispositivo físico, hemos de comprobar que la aplicación responde correctamente en todas las situaciones posibles.

Para ello, he distribuido la aplicación sobre 5 iPhones diferentes, con el fin de comprobar que la aplicación funciona correctamente en distintos dispositivos. Además de esta forma también puedo comprobar el correcto funcionamiento de las notificaciones Push.

La finalidad de las pruebas en dispositivos finales es comprobar que la aplicación va a responder correctamente en cualquier situación, por ello se somete a la aplicación a procesos de estrés, haciendo transiciones de estado rápidas o limitando la conectividad del dispositivo.

Se ha distribuido sobre cuatro iPhone4 y un iPhone 3GS, de los cuales tres tienen instalada la versión iOS 4, mientras que los otros dos están actualizados a la versión iOS 5.

Los tests aplicados a cada uno de ellos han resultado satisfactorios. Todos los dispositivos respondían por igual ofreciendo una experiencia de usuario acorde a las expectativas.

En situaciones de conectividad WiFi, la aplicación carga en un tiempo inapreciable, lo que dota a la aplicación de una disponibilidad máxima.

En situaciones donde la cobertura es 3G o EDGE, los tiempos de carga son mínimos y no perjudican en exceso la experiencia de usuario.

En caso de estar utilizando conectividad 3G o EDGE es recomendable dejar el WiFi activado, ya que aunque no se utilice para la transmisión de información, si que es un elemento importante a la hora de obtener la localización del dispositivo. La posición inicial del dispositivo se puede conseguir mediante una estimación aproximada utilizando triangulación de las señales provenientes de las estaciones de telefonía. Este método tiene unos niveles de precisión más bajos que la triangulación de señales WiFi, que basa su funcionamiento en las mismas premisas que la triangulación GSM, pero al haber muchas más fuentes de información, es decir, más puntos de acceso, el margen de error disminuye considerablemente.

Estos recursos son utilizados al inicio de la localización. Una vez se ha establecido conexión con el satélite GPS, los servicios de localización asistida ya no son necesarios, ya que la precisión que ofrece la tecnología GPS es mucho mayor.

Hemos comprobado que a diferentes niveles de carga, la aplicación ofrece unas prestaciones similares. Hemos aumentado el tamaño de los ficheros XML hasta límites que jamás serán utilizados, aun así, la aplicación es capaz de realizar las operaciones necesarias en unos tiempos más que aceptables.

Por lo que a falta de pequeños cambios en la interfaz como puede ser el icono de la aplicación, la imagen de inicio (Splash Screen) o completar los ficheros XML con la información correspondiente, podemos afirmar que la aplicación está lista para su distribución.

6. Conclusiones

En este trabajo de la Tesis de Master se ha implementado una aplicación distribuida utilizando técnicas de localización, posicionamiento, realidad aumentada y notificaciones Push.

El propósito de la aplicación es proporcionar al usuario toda la información que necesita de una forma rápida, sencilla e intuitiva. Una vez concluido el desarrollo de la aplicación podemos afirmar que los objetivos previstos inicialmente se han cumplido satisfactoriamente.

El desarrollo de este proyecto me ha permitido además entender cuales son los procesos que hay que cumplimentar para el desarrollo de aplicaciones para dispositivos móviles.

Experiencia en el desarrollo

Unas de las frustraciones más comunes en los desarrolladores al inicio del proceso de aprendizaje, es ver que los resultados no ofrecen una aplicación práctica real, y hasta que se consigue dominar la programación lo suficiente como para desarrollar aplicaciones útiles, puede pasar bastante tiempo. Considero que el entorno que ha confeccionado Apple contribuye a acelerar este proceso.

Personalmente, me encuentro satisfecho con el trabajo desarrollado y considero que tengo los fundamentos suficientes para la realización de aplicaciones para dispositivos iOS.

Una vez concluido el desarrollo, uno se da cuenta de que el esfuerzo que se requiere durante el aprendizaje no es excesivo y las limitaciones residen principalmente en la imaginación del desarrollador. Resulta satisfactorio ver como es posible implementar con facilidad prácticamente cualquier idea, en parte gracias a las facilidades del entorno.

Comparativa de los Sistemas Operativos Móviles

Para la realización del proyecto he tenido que analizar con profundidad los diferentes sistemas operativos móviles existentes en el mercado. Me ha permitido conocer las plataformas de desarrollo que rodean cada uno de los sistemas operativos y tener una mejor perspectiva de cada uno de ellos.

Como conclusión me quedo con que en la actualidad el sistema operativo con una tienda de aplicaciones mejor establecida es iOS. Esto no quiere decir que en un futuro vaya a ser así, pero en la actualidad lo es. Es por ello que la plataforma generalmente más rentable para los desarrolladores es iOS.

Ahora bien, desde el punto de vista del desarrollador, lo recomendable es desarrollar para el mayor número de plataformas copando el mayor porcentaje de mercado posible.

Mercado Laboral

La percepción que tengo respecto al mercado laboral en torno al desarrollo de dispositivos móviles es muy positiva. Consultando ofertas de trabajo en diferentes portales web, uno se da

cuenta de que existe una demanda grande por parte de las empresas en busca de desarrolladores para dispositivos móviles.

Esta percepción se ve sustentada por las opiniones de los ponentes del seminario "Desarrollando aplicaciones móviles: retos y oportunidades profesionales" organizado por la Escuela Técnica Superior de Ingeniería Informática el día 30 de noviembre.

En la actualidad la demanda está encabezada por desarrolladores iOS, seguidos de Android y HTML5. En la mayoría de los casos, se exige ciertos niveles de conocimiento de SQL y PHP, algo totalmente lógico dada la vinculación existente entre los dispositivos móviles y la web.

Posibilidad de venta de la aplicación

En la actualidad me encuentro en negociaciones con varios entes que se han mostrado interesados en la aplicación.

En un primer momento pensaba distribuir la aplicación por mi mismo, pero pensé que quizás la aplicación podría ser un buen medio publicitario para empresas del sector turístico, ya que es una forma muy interesante de llegar al público y ofrecer un servicio bajo la imagen corporativa de la empresa.

Además, quizás estas empresas tenga más facilidad a la hora de contactar con los posibles anunciantes de la pestaña de Servicios, ya que poseen los contactos necesarios ya que es su campo de trabajo. Por ello, pienso que podría ser un modelo de negocio beneficioso para ambas partes.

Aprendido un nuevo lenguaje

Durante el desarrollo del proyecto me he visto forzado a aprender un lenguaje relativamente distinto al resto de los lenguajes que conocía, especialmente en la sintaxis. Objective-C resulta un lenguaje fácil de comprender y tiene una curva de aprendizaje corta.

Además todo el entorno que ofrece Apple ayuda a que el programador se sienta cómodo, tanto la documentación como el IDE están plenamente integrados y facilitan las tareas del desarrollador.

Base de código extrapolable

Al ser una aplicación distribuida, los datos que maneja la aplicación no residen en la aplicación en sí, sino que esta información es consultada a través de servicios web. De modo que el código de la aplicación tiene cierto grado de independencia respecto a los datos, lo que me puede permitir la creación de nuevas aplicaciones basadas en un esquema parecido, cambiando únicamente elementos de la interfaz gráfica y los ficheros XML a leer.

De esta forma, la aplicación Fallas proporciona el código base necesario para crear aplicaciones similares con otra temática como podría ser, un listado de gasolineras, una guía para el campus universitario, una aplicación temática para otras festividades o un listado de las tiendas de una franquicia, por poner algunos ejemplos.

7. Bibliografía

- **Desarrollo de aplicaciones para iPhone**, John Ray, Sean Johnson. SAMS
- **Programming iOS 4**, Matt Neburg. O'REILLY
- **Beginning iPhone 4 Development Exploring the iOS SDK**, Dave Mark, Jack Nutting, Jeff LaMarche, APRESS
- **iPhone SDK Essential Training**, Lynda.com
- **Apple Documentation**, developer.apple.com
- **Stackoverflow**, stackoverflow.com
- **MacRumors Forums**, macrumors.com/forums
- **iDev Recipes**, idevrecipes.com
- **Github social coding**, github.com
- **Layar Player**, layar.com
- **QCAR Qualcomm**, https://developer.qualcomm.com/develop/mobile-technologies/augmented-reality
- **3DAR**, http://3dar.us/
- **ARKit original**, https://github.com/zac/iphonearkit/
- **ARKit fork**, https://github.com/nielswh/iPhone-AR-Toolkit
- **Urban Airship**, http://urbanairship.com/
- **Flurry**, http://www.flurry.com/