ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Aplicación para dispositivos móviles Android: Guía de los edificios de la Universidad Politécnica de Cartagena

Autor: Luis Gonzalo Soto Aboal
Director: Juan Ángel Pastor Franco

Agosto 2011

Autor	Luis Gonzalo Soto Aboal
Correo electrónico del autor	luisgonzalosotoaboal@gmail.com
Director	Juan Ángel Pastor Franco
Correo electrónico del director	juanangel.pastor@upct.es
Título del PFC	Aplicación para dispositivos móviles Android: Guía de los edificios de la Universidad Politécnica de Cartagena

Resumen:
El proyecto consiste en el desarrollo de una aplicación para dispositivos móviles que utilizan el sistema operativo Android (smartphones y tablets).
Mediante esta aplicación, GUÍA UPCT, el usuario podrá utilizar su móvil para ver en un mapa la localización de los edificios que componen el campus de la Universidad Politécnica de Cartagena, calcular la ruta a pie más corta para llegar a los mismos desde su posición, consultar planos de los edificios en los cuales se mostrarán puntos de interés, y más información útil sobre cada uno.
A lo largo de la memoria se explicará el contexto de las aplicaciones móviles y del sistema Android, y todos los conceptos y herramientas aplicados durante el desarrollo, de cara a resultar lo más didáctico posible.
La memoria en sí se presenta en un formato de páginas adecuado para su impresión y encuadernación.

Titulación	Ingeniería Técnica de Telecomunicación
Intensificación	Telemática
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de presentación	Mayo de 2012

A Lorena, mi pareja, mi alma gemela,
por sus ideas y por apoyarme siempre.
Y a la memoria de mi abuelo Gonzalo.

Índice de contenido

Introducción

Planteamiento inicial

Uno de los aspectos destacables de la Universidad de Cartagena es que su campus no es un recinto cerrado de edificios, sino que está integrado en la propia ciudad. Combinando la construcción de nuevas y modernas instalaciones (como el campus de Alfonso XIII en su momento, o el nuevo edificio de I+D+I) con la rehabilitación de edificios emblemáticos del casco antiguo (como el CIM, La Milagrosa, Antigones o el Club Santiago), la Universidad consigue ser una parte esencial de la ciudad, y crecer con ella. Podría decirse que el campus de la Universidad, es Cartagena en sí misma.

Sin embargo, para un alumno recién llegado, esta dispersión de edificios puede dificultar en parte la adaptación durante los primeros meses. Una futura Ingeniera Industrial que quiera ir a estudiar a los aularios del paseo Alfonso XIII, un estudiante de Turismo que quiera acercarse al Servicio de Idiomas en el CIM, o un grupo de compañeros de 1º de Telemática que estén buscando el ALA de Industriales, seguramente tengan que preguntar un par de veces antes de llegar a su destino.

Por otra parte, en el mundo actual, nos encontramos con un escenario tecnológico en el que la estrella es la movilidad. El teléfono móvil sustituye al fijo, los ordenadores de sobremesa son reemplazados por ligeros portátiles, e incluso estos últimos empiezan a perder su puesto a favor de dispositivos de tipo "teléfono inteligente" (*smartphone*) que encierran, en la palma de nuestra mano, una capacidad de procesamiento que hace apenas 10 años no podíamos sino soñar. El negocio de las aplicaciones para móviles está en pleno auge, y los juegos y contenidos multimedia que se pueden disfrutar en este tipo de dispositivos no tienen nada que envidiar a los que ofrecen plataformas más clásicas.

Esta ubicuidad de la información comienza a tener, además, cierto reflejo en la forma de desarrollar aplicaciones: en un mundo en el que todos estamos interconectados, en el que todos los servicios están disponibles en cualquier momento para todo el mundo ¿por qué reinventar aplicaciones? ¿Por qué no usar las aplicaciones que ya existen para añadirles valor? Este nuevo tipo de aplicaciones, denominadas en inglés *mash-ups*, está teniendo mucho éxito y demuestra que para crear aplicaciones que generen valor al usuario, no siempre es necesario empezar de cero ni reinventar la rueda: basta con saber agrupar los servicios adecuados de manera que juntos formen una aplicación con mayor utilidad que la que tendrían cada uno por separado.

Surge así la idea de crear una aplicación que, ejecutada en un terminal móvil e integrando diferentes servicios, permita a cualquier alumno y en cualquier parte de la ciudad: localizar rápidamente en un mapa cualquier edificio de la Universidad; consultar la mejor ruta para llegar a cualquiera de ellos; obtener información de las instalaciones y puntos de interés (POI, por sus siglas en inglés) de cada edificio, como

podrían ser: cafetería, ALA, secretaría, biblioteca o reprografía; e incluso acceder a información adicional vía Internet.

Objetivos

La creación de una aplicación para dispositivos móviles, que permita:

- Consultar en un mapa la localización de los principales edificios del campus de la UPCT.
- Visualizar en dicho mapa la ruta más corta para llegar a pie hasta dicho edificio.
- Mostrar un plano de cualquiera de los edificios del mapa, en el que vendrán indicados los puntos de interés de dicho edificio, y éstos al ser seleccionados mostrarán información adicional sobre el punto de interés en cuestión.
- Mostrar una ficha de cada edificio con información general del mismo, descripción y posibilidad de acceso a la página de la Escuela que alberga.

La aplicación deberá crearse usando lenguaje Java, para funcionar sobre el sistema operativo Android y hacer uso de sus capacidades para integrar diferentes API y servicios ya existentes como por ejemplo Google Maps, localización basada en GPS y redes móviles, etc.

Por último, la aplicación deberá diseñarse también de manera que permita ilustrar la aplicación de patrones conocidos de diseño *software*, especialmente el patrón MVC por ser éste utilizado de manera didáctica en algunas de las asignaturas de la carrera, además de ser muy usado en aplicaciones reales.

Requisitos

Código de requisito	*REQ01*	**Prioridad**	*Alta / Imprescindible*
Nombre	*Mapa de edificios con Google Maps*		
Descripción	*La aplicación mostrará un mapa satélite de la zona de Cartagena, en el cual se indicará la ubicación de los principales edificios que conforman el campus de la Universidad Politécnica de Cartagena.* *Dicho mapa hará uso de las librerías y servicios de Google Maps, por ser éste un servicio de fiabilidad contrastada y además gratuito. Tendrá así mismo los controles de zoom y desplazamiento encontrados habitualmente en aplicaciones similares.* *El usuario podrá pulsar sobre cada edificio para obtener una lista o diálogo con las acciones disponibles.*		

Código de requisito	*REQ02*	**Prioridad**	*Alta / Imprescindible*
Nombre	*Posición actual del dispositivo móvil*		
Descripción	*La aplicación dispondrá de la lógica necesaria para obtener o calcular la posición geográfica actual del dispositivo con una precisión suficiente, indicando al usuario dicha posición en el mapa mediante un icono distinto del de los edificios.* *Esta geolocalización se llevará a cabo utilizando los recursos de los que disponga el terminal móvil (que dependiendo del modelo podrán incluir: GPS, WiFi, GPRS/3G) y, de entre ellos, siempre el de mayor precisión. Dado que algunos de estos recursos pueden ser deshabilitados a voluntad por el usuario del dispositivo, la aplicación se limitará a utilizar los recursos disponibles, no habilitando en caso necesario ninguno de ellos sin la autorización explícita del usuario.*		

Código de requisito	*REQ03*	**Prioridad**	*Alta / Imprescindible*
Nombre	*Ruta hasta un edificio*		
Descripción	*Al pulsar sobre un edificio concreto, la aplicación ofrecerá al usuario la opción de mostrar la ruta a pie desde la ubicación actual del dispositivo hasta el edificio señalado. Esta ruta se*		

	pintará sobre el mapa con un color distintivo y será independiente de los cambios de posición del usuario (no tendrá funcionalidades de navegación).

Código de requisito	*REQ04*	**Prioridad**	*Alta / Imprescindible*
Nombre	*Plano de un edificio*		
Descripción	*La aplicación permitirá al usuario visualizar un plano del edificio seleccionado. Sobre dicho plano el usuario tendrá, al igual que en el mapa, la capacidad de hacer zoom y desplazamientos.* *El plano será sencillo (no arquitectónico) y permitirá identificar fácilmente los accesos al edificio y estructuras principales.*		

Código de requisito	*REQ05*	**Prioridad**	*Alta / Imprescindible*
Nombre	*Puntos de interés (POI) de un edificio*		
Descripción	*Cada plano de edificio deberá marcar, ya sea directamente o a petición, la posición dentro del edificio de una serie de servicios que se consideran de interés para el usuario. Ejemplos de dichos Puntos de Interés son: cafetería, biblioteca, reprografía, secretaría, A.L.A., salón de actos, oficinas de servicios al estudiante (COIE/SEEU, servicio de actividades deportivas, servicio de idiomas), recepción...* *Los puntos se mostrarán sobre el plano mediante una marca de color destacado, de tamaño suficiente para que el usuario la pueda pulsar con precisión, y acompañados del nombre del servicio.*		

Código de requisito	*REQ06*	**Prioridad**	*Media / Deseable*
Nombre	*Información sobre POI*		
Descripción	*Al pulsar sobre cada uno de los Puntos de Interés de un plano, se mostrará al usuario un pequeño diálogo con información sobre el mismo: nombre del servicio y descripción del servicio. Además dicho diálogo ofrecerá la opción de visitar una página web con información ampliada.* *Por ejemplo, al pulsar en la cafetería se mostraría el precio del menú del día, y en la página web se podrían consultar el horario, los menús de la semana y el resto de platos y precios.*		

Código de requisito	*REQ07*	**Prioridad**	*Alta / Imprescindible*
Nombre	*Listado de edificios*		
Descripción	*Dado que inicialmente el usuario no tiene por qué saber cuál de todos los edificios del mapa se corresponde con el que él quiere visitar / consultar, se hace necesaria la implementación de una funcionalidad de listado de los edificios existentes, de manera que el usuario pueda identificar el edificio deseado rápidamente por nombre o escuela.* *La lista de edificios mostrará, como mínimo:* • *Pequeña imagen del edificio.* • *Nombre del edificio (Antigones, La Milagrosa...)* • *Nombre de la escuela u órgano que alberga (ETSIT, Rectorado...)*		

Código de requisito	*REQ09*	**Prioridad**	*Baja / Opcional*
Nombre	*Búsqueda de edificios*		
Descripción	*Como ampliación de la funcionalidad de listado de edificios, se propone implementar sobre la misma una función de búsqueda de edificios mediante palabras clave. Por ejemplo: una búsqueda por "teleco" mostraría Antigones; una búsqueda por "ingeniería" mostraría las escuelas de ingeniería, pero no el Rectorado ni el CIM; "biblioteca" mostraría sólo Antigones y la biblioteca del campus de Alfonso XIII; etc.*		

Código de requisito	*REQ10*	**Prioridad**	*Media / Deseable*
Nombre	*Ayuda de la aplicación*		
Descripción	*Aunque la aplicación pretende ser sencilla y bastante intuitiva, se considera adecuada la inclusión de una pequeña pantalla de ayuda para el usuario, en formato texto o HTML.* *El contenido de la ayuda será el siguiente:* • *Descripción de la aplicación* • *Controles básicos* • *Funcionalidad de cada pantalla/actividad* • *Requisitos de funcionamiento (conexión a internet...)* *En esta pantalla de ayuda se podrán incluir enlaces adicionales que amplíen la ayuda o den acceso a información suplementaria que se considere útil.*		

Código de requisito	*REQ11*	**Prioridad**	*Media / Deseable*
Nombre	*Pantalla de inicio*		

Descripción	*La aplicación mostrará siempre, como pantalla inicial, una actividad con las siguientes características:*
	• *Logotipo de la aplicación.*
	• *Nombre de la aplicación.*
	• *Acceso a las funcionalidades principales:*
	  ○ *Mapa*
	  ○ *Listado*
	  ○ *Ayuda*
	El objetivo de esta pantalla de inicio, además de ofrecer al usuario la elección de funcionalidad, es no dejar al usuario esperando mientras se llevan a cabo las tareas iniciales de recogida de datos y comprobación de conectividad y permisos.		

Código de requisito	*REQ12*	**Prioridad**	*Media / Deseable*
Nombre	*Funcionalidad "offline"*		
Descripción	*Se considera deseable que parte de los datos que muestra la aplicación estén disponibles de forma offline, es decir, sin necesidad de tener habilitada una conexión de datos para consultarlos; evitando de esta manera en lo posible que el uso de la aplicación genere un gasto económico al usuario.* *Los datos que se deben poder acceder de forma offline serán:* • *Listado de edificios* • *Información básica del edificio (nombre, imagen, escuela/órgano, descripción)* • *Plano y puntos de interés del edificio* *Se entiende que las funcionalidades de mapa y ruta, por hacer uso de las librerías y servicios de Google Maps, necesitan indispensablemente de una conexión de datos y quedan excluidas de este requisito.*		

Código de requisito	*REQ13*	**Prioridad**	*Media / Deseable*
Nombre	*Ficha de edificio*		
Descripción	*Se plantea la creación de una pantalla adicional que presente al usuario toda la información de un edificio (imagen, nombre, escuela, descripción, plano), a modo de ficha informativa.* *Dicha pantalla será accesible desde el listado de edificios, permitiendo al usuario consultar toda la información disponible*		

	en modo fuera de línea sin necesidad de acceder al mapa de edificios.

Código de requisito	*REQ14*	**Prioridad**	*Alta / Imprescindible*
Nombre	*Usabilidad y robustez*		
Descripción	*Será obligatorio que la aplicación sea mínimamente robusta y no presente errores, reinicios ni bloqueos durante un uso "normal" de la misma en un dispositivo Android. Este uso normal incluye situaciones como: cambios de orientación de la pantalla del dispositivo, de horizontal a vertical y viceversa; paso de la aplicación a segundo plano por llamadas entrantes, alarmas o cualesquiera otras aplicaciones similares; imposibilidad de determinar la ubicación del dispositivo durante el cálculo de la ruta a un edificio, etc. En todas ellas la aplicación deberá mantener su estado y continuar con la ejecución de manera normal siempre que sea posible, mostrando mensajes controlados de error cuando no lo sea.*		

Casos de Uso

A continuación se detallan los casos de uso más representativos de la aplicación, los cuales cubren prácticamente toda la funcionalidad disponible.

Caso de uso 1: Visualización de ruta

Descripción	
Muestra los pasos que sigue un usuario que quiere consultar visualmente sobre el mapa, la ruta a pie desde su ubicación actual hasta un edificio de la UPCT de su elección.	
Flujo principal	
1. El usuario arranca la aplicación.	2. El sistema muestra la pantalla de Inicio de la aplicación, con las opciones "Mapa", "Listado" y "Ayuda".
3. El usuario selecciona la opción "Mapa".	4. El sistema muestra un mapa satélite, marcando la posición de cada edificio con un icono en forma de edificio. 5. El sistema obtiene la posición del propio usuario y la muestra sobre el mismo mapa con un marcador parpadeante. *{Ver flujo alternativo 1}*
6. El usuario pulsa sobre el icono del edificio al que quiere ir.	7. El sistema muestra un diálogo con las opciones "Ver ruta", "Ver plano" y "Cancelar".
8. El usuario pulsa sobre la opción "Ver ruta".	9. El sistema obtiene la ubicación actual del usuario y dibuja sobre el mapa el trazado de la ruta más óptima a pie, desde ahí hasta el edificio seleccionado. *{Ver flujo alternativo 2}* 10. El sistema centra el mapa en el punto de partida de la ruta (posición del usuario).
Flujos alternativos	
Flujo alternativo 1	
3. El usuario selecciona la opción "Mapa".	4. El sistema muestra un mapa satélite, marcando la posición de cada edificio con un icono en forma de edificio. 5. El sistema no consigue obtener la posición del propio usuario y por tanto ésta no se muestra en el mapa.
Flujo alternativo 2	

8. El usuario pulsa sobre la opción "Ver ruta".	9. El sistema no puede obtener la posición actual del usuario. 10. El sistema muestra un mensaje de error indicando que no se puede obtener la posición del usuario, y no pinta ninguna ruta.

Caso de uso 2: Información ampliada sobre Punto de Interés

Descripción	
Muestra los pasos que sigue un usuario que quiere consultar información ampliada sobre un punto de interés (POI) de uno de los edificios de la Universidad.	
Flujo principal	
1. El usuario arranca la aplicación.	2. El sistema muestra la pantalla de Inicio de la aplicación, con las opciones "Mapa", "Listado", "Ayuda".
3. El usuario pulsa sobre la opción "Listado".	4. El sistema muestra un listado de los edificios disponibles en la aplicación, incluyendo para cada uno de ellos una imagen descriptiva, el nombre del edificio y el nombre de la entidad que lo utiliza.
5. El usuario pulsa sobre un edificio de la lista.	6. El sistema muestra una pantalla con información ampliada del edificio y las opciones "Mapa", "Plano", "Visitar web".
7. El usuario pulsa sobre la opción "Plano".	8. El sistema muestra el plano del edificio y, sobre el mismo, una serie de puntos rojos representando cada uno de los POI definidos para el edificio.
9. El usuario pulsa sobre el POI del cual quiere obtener información.	10. El sistema muestra un diálogo con información resumida del POI y las opciones "Visitar web", "Cancelar".
11. El usuario pulsa sobre la opción "Visitar web".	12. El sistema inicia el navegador web predeterminado y accede a la URL de la página web asociada al POI seleccionado, mostrando la información contenida en dicha web.
Flujos alternativos	
No aplican.	

Aplicaciones móviles

Introducción

El mundo de las aplicaciones móviles está viviendo hoy en día, sin duda, su momento más dulce. Puede que sea por el avance tecnológico de los dispositivos móviles, especialmente los *smartphones*, que ha conseguido equiparar la capacidad gráfica y de procesamiento de éstos a la de los ordenadores que podemos encontrar hoy día en muchos hogares; o quizá sea porque este mismo avance ha permitido que tecnologías como GPS y WiFi, que hace menos de una década sólo podían encontrarse en aparatos destinados al hogar o en productos dirigidos a mercados muy específicos (navegadores GPS), se integren ahora "de serie" en casi cualquier teléfono móvil, multiplicando las posibilidades de uso. O tal vez sea gracias al auge de las redes sociales, de la necesidad de información inmediata, del qué estás haciendo y del qué está pasando; de la cultura, en fin, del "aquí y ahora", que lo inunda todo con sus torrentes de actualizaciones, noticias, eventos y datos que parecen destinados a ser enviados y consumidos desde dispositivos móviles porque, si esperas a llegar a casa, habrán desaparecido enterrados bajo miles de cosas más recientes. Lo más probable es que sea una combinación de estos tres factores y muchos otros.

Figura 1. Un portátil Apple Powerbook G4 de 2005 (izquierda) tiene la misma velocidad de procesador, cantidad de memoria RAM y espacio de almacenamiento que un teléfono móvil Samsung Galaxy S2 de 2011 (derecha).

En cualquier caso, nos encontramos actualmente con un mercado, el de las aplicaciones móviles, que ha ido creciendo y evolucionando de forma pareja a como lo ha hecho el de los dispositivos móviles. Ha pasado de ser casi inexistente, a estar orientado puramente al entretenimiento, hasta encontrar hoy día nichos de mercado en prácticamente todos los aspectos de nuestra vida: lectores de documentos para la oficina, aplicaciones para crear música, lectores de libros electrónicos, juegos sencillos o completos simuladores en 3D, aplicaciones para ir de compras, para navegación mediante GPS, para convertir el teléfono móvil en un punto de acceso WiFi, aplicaciones para leer la Biblia, para calcular los transbordos en el metro, para

retoque fotográfico, aplicaciones para leer los periódicos extranjeros, para consultar los síntomas de una enfermedad... Aplicaciones todas ellas, que toman ventaja de las velocidades de transmisión de las redes actuales, de la posibilidad de conocer la localización del usuario, de la capacidad de proceso de los nuevos modelos y de la ubicuidad del acceso a internet y a toda la información que alberga, para ofrecer al usuario la que necesita en en ese momento.

Definición y orígenes

Entendemos como aplicación móvil, cualquier aplicación *software* creada por terceros y destinada a su instalación y ejecución en un dispositivo móvil, esto es de tamaño reducido e ideado para ser utilizado de manera inalámbrica.
Cuando a mediados de la década de 1990 aparecieron los primeros dispositivos móviles que se comercializaron con éxito, las famosas PDA (del inglés *Personal Digital Assistant*), el mercado de las aplicaciones móviles era muy discreto. Estos dispositivos —destinados casi siempre a profesionales en continuo movimiento, como ejecutivos y comerciales, o relacionados con el mundo de la tecnología— estaban muy limitados tecnológicamente, y de hecho los primeros modelos requerían de una "estación base" conectada a otro equipo para poder transferir y almacenar la información en dicho equipo, dado que las PDA no tenían memoria de almacenamiento propia.

Figura 2. Dispositivo PDA, modelo Palm III de 1999, desarrollado por Palm Computing. Pantalla táctil LCD con 4 tonos de gris, 8 MB de memoria RAM. La imagen muestra el dispositivo acoplado a su base de conexión al PC.

Además, las aplicaciones que el fabricante incluía por defecto: calendario, agenda de contactos, pequeños editores de texto y,más adelante, clientes de correo y navegadores web, eran más que suficientes para la utilización que se les daba. Esto suponía que las aplicaciones móviles existentes fueran desarrolladas casi siempre a petición de alguna empresa privada, para su utilización propia o, como mucho, ampliaciones y mejoras de las aplicaciones ya instaladas, dado que la tecnología del dispositivo no daba para más y, por tanto, no merecía la pena desarrollar aplicaciones adicionales. Aunque la evolución posterior propició que comenzaran a aparecer

aplicaciones desarrolladas por terceros, el mercado ya había sido conquistado por los teléfonos móviles.

Telefonía móvil: J2ME y WAP

Lo que no sucedió con las PDA, sucedió con los teléfonos móviles: en la segunda mitad de los 90, la introducción de las tecnologías GSM y EDGE con su sistema de mensajes cortos SMS, el protocolo WAP, el abaratamiento de costes y la liberalización del mercado de operadores se unieron para provocar un *boom* en la utilización de telefonía móvil, pasando sólo en España de 1 millón de usuarios en 1996 a casi 30 millones en 2001.

Aunque los primeros terminales de 2ª generación (GSM) se limitaban a ofrecer la funcionalidad básica de comunicación por voz, envío de mensajes SMS y listín telefónico —con pantallas monocromas de dos o tres líneas de texto y, en muchos casos, ni siquiera letras minúsculas; aunque había algunas excepciones como el Nokia 9000 Communicator que, con sus aplicaciones de correo y navegación web, sólo estaba al alcance de los bolsillos más pudientes—, ya a finales de los 90 empezaron a aparecer los primeros móviles destinados al público general que incorporaban aplicaciones sencillas de tipo calendario, reloj y agenda, e incluso juegos. Sin embargo, en todos los casos eran aplicaciones propietarias incluidas por los propios fabricantes; los terminales de entonces no tenían ningún mecanismo sencillo que permitiera la instalación de aplicaciones de terceros.

Figura 3. De izquierda a derecha: en 1997, el modelo de Alcatel "One Touch Easy", con sus dos líneas de texto, no permitía margen para aplicaciones; en 1998, el modelo "5110" de Nokia es uno de los primeros en incorporar

aplicaciones de entretenimiento como el juego "Snake"; en 2002, el Nokia "7210" ya venía con WAP y J2ME y las posibilidades se disparaban.

El éxito de los terminales que incorporaban juegos y aplicaciones a su funcionalidad predeterminada, provocó que los fabricantes reorientaran parte de sus líneas de producción a crear dispositivos orientados un poco más hacia el entretenimiento. Un cambio de mentalidad que abrió el mercado de la telefonía a un público joven que exigía constantemente más novedades, más tecnología, más variedad, más posibilidades, y que hizo avanzar el mercado a un ritmo vertiginoso. Con el cambio de siglo llegaron al público general nuevos terminales, más pequeños, con más capacidad y con pantallas de miles de colores, algunos de los cuales incorporaban dos tecnologías que, juntas, formaban el caldo de cultivo perfecto para el mercado de las aplicaciones: J2ME y WAP.

Java 2 Mobile Edition (J2ME, hoy conocido como JME) surgió en 2001 como un subconjunto básico de las librerías y API de Java, diseñado para ser ejecutado en dispositivos embebidos como sistemas de control industrial, pequeños electrodomésticos y, por supuesto, terminales móviles y PDA. Su llegada al mundo de lo inalámbrico facilitó enormemente el desarrollo de aplicaciones para este tipo de dispositivos, puesto que ya no era necesario conocer lenguajes específicos de cada dispositivo o fabricante ni disponer obligatoriamente de un terminal en el que hacer las pruebas: los programadores podían crear sus aplicaciones cómodamente en su ordenador personal, con un emulador y usando un lenguaje fácil y potente como Java. J2ME abrió las puertas de los terminales móviles a los desarrolladores.

WAP (Wireless Application Protocol), por otra parte, es un conjunto de protocolos que facilita la interoperabilidad de dispositivos móviles en redes inalámbricas como GSM y CDMA. Aunque comenzó a desarrollarse en 1997, fue a partir de 2001 cuando empezó a extenderse su uso, con la introducción de la funcionalidad "WAP Push". Esta nueva característica permitía a los operadores enviar a sus usuarios notificaciones con enlaces a direcciones WAP, de manera que el usuario sólo tenía que activarlas para acceder a contenidos en red. Gracias a esto, los terminales móviles podían intercambiar información con servidores web y descargar aplicaciones específicas de una manera sencilla y en cualquier parte. El protocolo WAP rompía así las barreras entre los usuarios de dispositivos móviles y los proveedores de contenidos.

Figura 4. Fragmento de una página de revista, anunciando multitud de contenidos multimedia accesibles mediante WAP y mensajes premium.

Facilidad de desarrollo y facilidad de distribución formaron una combinación exitosa (no sería la última vez, como veremos más adelante) e hicieron surgir con verdadera fuerza el mercado de las aplicaciones móviles. Los desarrolladores creaban juegos y aplicaciones en Java que servían para múltiples modelos; los proveedores anunciaban estas aplicaciones en revistas especializadas, periódicos e incluso en televisión; y los usuarios descargaban en sus teléfonos fondos de pantalla, tonos y juegos, mediante el simple envío de mensajes SMS a números de tipo *premium* (tarificados de manera especial por la operadora para ofrecerle un beneficio a la compañía que distribuye las aplicaciones).

No obstante, pronto el escenario volvería a cambiar: se estaba preparando la llegada de los *smartphones*.

Symbian

En el año 2000, la compañía Ericsson provocó un salto cualitativo en el mercado de los dispositivos móviles: presentó su modelo Ericsson R380 Smartphone, el primer teléfono móvil con pantalla táctil y el primero que integraba toda la funcionalidad de una auténtica PDA manteniendo al mismo tiempo el tamaño, peso y diseño de los terminales de entonces. Aunque la interfaz se parecía demasiado a la de las PDA de la época y a pesar de que era un modelo "cerrado" que no permitía instalar aplicaciones de terceros, fue muy bien recibido por el público más técnico.

Figura 5. Teléfono móvil Ericsson modelo R380 (2000). Fue el primero en combinar un teléfono móvil con las funcionalidades de una PDA.

Este avance fue gracias a que el R380 usaba una versión de un sistema operativo, EPOC, que ya utilizaban algunas PDA de la época y que fue adaptado para su uso en terminales móviles. Dicha versión, conocida como Symbian, comenzó a desarrollarse en 1998 por una asociación conjunta de los fabricantes Nokia, Ericsson, Motorola y Psion, que tenían como objetivo aprovechar la incipiente convergencia entre teléfonos móviles y PDA para desarrollar un sistema operativo abierto que facilitara el diseño de los futuros dispositivos. El Ericsson R380 fue el primer modelo que salió al mercado con este sistema operativo y, ante el éxito cosechado y aprovechando el progreso de la tecnología utilizada en los terminales más recientes, Nokia y Ericsson enfocaron parte de sus esfuerzos a suplir las carencias de Symbian.

Nokia, comprendiendo el potencial del nuevo sistema operativo, decidió utilizarlo en sus modelos "Communicator" que, a pesar de ser los más avanzados (y caros) de toda su gama —ya desde varios años antes incorporaban acceso a Internet, correo, un teclado completo y varias otras aplicaciones—, estaban hechos sobre una plataforma que empezaba a quedarse pequeña.

Con su modelo Communicator 9210, Nokia introdujo a mediados del año 2001 en el mercado la versión 6 de Symbian OS, con multitarea real y acompañado de la nueva interfaz gráfica "Series 80" a todo color. El modelo fue muy bien recibido en algunos sectores de público que no estaban familiarizados con las PDA y para los cuales el Communicator suponía una novedosa solución "todo en uno" para sus negocios.

Figura 6. Teléfono móvil Nokia, modelo "Communicator" 9210 (2001), con teclado incorporado. Fue el primero en incorporar Symbian 6.0 y la interfaz "Series 80".

Sin embargo, el 9210 carecía de pantalla táctil y esto fue visto como un paso atrás por muchos clientes, que estaban acostumbrados a las pantallas táctiles de las PDA y que esperaban que un modelo con características de PDA como era el 9210 también tuviera una. Además, su tamaño de "ladrillo" lo dejaba fuera de la tendencia a la reducción de tamaño que estaba experimentando el mercado generalista de dispositivos.

Nokia no estaba dispuesto a tirar la toalla y en el segundo cuarto de 2002 puso sobre la mesa su modelo 7650: el primer *smartphone* auténtico, con cámara incorporada y con un diseño agradable y tamaño similar al del resto de teléfonos móviles del mercado. Traía Symbian OS 6.1 y una interfaz gráfica, "Series 60", diseñada específicamente para terminales con pantalla y teclado "cuadrados", que era el resultado del trabajo de Nokia sobre la anterior "Series 80" para facilitar al máximo su uso. Y lo más importante de todo: permitía instalar aplicaciones de terceros.

Figura 7. Teléfono móvil Nokia 7650 (2002). Con Symbian OS 6.1 y 3,6 MB de espacio adicional, permitían instalar aplicaciones como por ejemplo una versión del navegador Opera.

Este modelo de Nokia fue un éxito instantáneo y el culpable, en gran medida, de que el mercado de las aplicaciones móviles se llenara de programas compatibles con dispositivos basados en S60. La posibilidad de instalar aplicaciones tanto nativas como basadas en J2ME, y de hacer cosas como comprobar el correo, hacer fotos o jugar a un juego, todo ello con una sola mano y utilizando un teléfono elegante que cabía cómodamente en el bolsillo, fue la mejor carta de presentación posible para la plataforma S60 de Nokia y de hecho, hasta 2010, S60 fue la interfaz más utilizada en todo el mercado de terminales móviles.

Sin embargo, todavía hacían falta un par de componentes más para desencadenar la siguiente explosión en el mercado de las aplicaciones móviles, y uno de ellos era el espacio disponible. El Nokia 7650, con sus 3,6 MB de espacio, apenas dejaba espacio para aplicaciones muy elaboradas, y ni siquiera un fichero MP3 de tamaño medio podía ser almacenado en él.

Ericsson, para entonces ya Sony Ericsson, se encargó de dar el siguiente paso en la evolución de Symbian, sacando al mercado su modelo P800. Este teléfono móvil fue el primero en traer Symbian 7.0, la siguiente versión del sistema operativo, así como una nueva interfaz gráfica desarrollada por Ericsson (antes de su fusión con Sony) y conocida como UIQ, que permitía instalar aplicaciones desarrolladas por terceros bajo C++ o bajo J2ME. El Sony Ericsson P800, además de su pantalla a todo color y su cámara VGA, llegaba acompañado de 16 MB de espacio pero podía, aprovechando la tecnología de almacenamiento mediante tarjetas SD desarrollada por Sony, ampliarse hasta unos increíbles (para entonces) 128 MB de espacio, lo que permitía al usuario utilizar su teléfono también como reproductor de música y hasta de vídeo.

Figura 8. Teléfono móvil Sony Ericsson P800 (finales de 2002), con teclado desmontable y pantalla táctil.

A partir de este momento, la evolución y expansión del sistema operativo Symbian sería, durante algunos años, imparable. Para los fabricantes, especialmente para Nokia y Sony Ericsson, era más fácil integrarlo en sus terminales al ser un sistema abierto; para los usuarios, era el sistema "de moda", el más compatible con todas las aplicaciones disponibles y el que más cosas podía hacer: varios modelos disponían de navegador web totalmente compatible con HTML, no solo WAP, y y también aparecieron modelos con funcionalidades y diseño ideados para actividades específicas como escuchar música o jugar a videojuegos, conquistando así cuota de mercado en nichos tradicionalmente apartados del mundo de la telefonía.

	2005	**2006**	**2007**	**2008**
Symbian	68,0%	76,1%	63,5%	52,4%
BlackBerry OS (RIM)	2,0%	9,2%	9,6%	16,6%
Windows (Microsoft)	4,0%	6,7%	12,0%	11,8%
iOS (Apple)	-	-	2,7%	8,2%
Android (Google)	-	-	-	0,5%
Otros (Linux, HP...)	28,0%	8,0%	12,1%	10,5%

Tabla 1. Cuota de mercado global de los principales sistemas operativos para terminales smartphone, de 2005 a 2008. Fuentes: Canalys, Gartner.

Y aunque los teléfonos más económicos siguieron utilizando sistemas cerrados

específicos para cada modelo (casi siempre compatibles con J2ME); y a pesar de la entrada en el mercado de otros competidores que daban un gran valor añadido al servicio de telefonía móvil —como es el caso del fabricante Research In Motion con sus dispositivos BlackBerry, de gran éxito en los países norteamericanos, que permitían mensajería instantánea y servicios de correo electrónico encriptado gracias a su red privada de servidores—; Symbian llegó a copar el mercado mundial de teléfonos móviles, con un 76% de cuota de mercado en 2006. Aún en 2008, más de la mitad de *smartphones* a la venta estaban basados en una versión u otra de Symbian.

Sin embargo, ya mediado 2007 aparecerían dos nuevos competidores que impactarían profundamente el monopolio de Symbian y cambiarían la forma de ver el mercado de las aplicaciones móviles: iPhone y Android.

IPhone y Android

Hemos empezado este capítulo hablando de las PDA. Hasta 1992, nadie había oído nunca hablar de una PDA. Sí, existían dispositivos portátiles que ofrecían una ayuda para organizar tareas y tomar notas; pero no fue hasta ese año que Apple acuñó el término *personal digital assistant (PDA)* para referirse a su última creación: el Apple MessagePad, la primera PDA auténtica, con un nuevo sistema operativo (Newton), pantalla táctil, reconocimiento de escritura y un aspecto e interfaz que serían después imitados por todos los modelos de PDA del mercado.

Hemos empezado este capítulo, por tanto, hablando de Apple; y es curioso, porque vamos a terminarlo hablando de Apple otra vez.

Apple, que a principios de los 90 había revolucionado el mercado de dispositivos móviles con la introducción de sus modelos basados en el sistema operativo Newton, no cosechó sin embargo el éxito que esperaba con ellos. Una autonomía muy limitada por el uso de pilas AAA, un sistema de reconocimiento de escritura defectuoso que requería demasiado tiempo de "aprendizaje" y la falta de conectividad inicial con el ordenador del usuario (los cables y software necesario se comercializaban aparte) fueron las principales causas de que, durante la década de los 90, el mercado fuera dominado por otras compañías con modelos más depurados y Apple se centrara más en su línea de ordenadores personales (llegando a cosechar enorme éxito a finales de siglo con su línea de ordenadores iMac y PowerBook y con su sistema operativo Mac OS).

Sin embargo, la mayor revolución de todas, la que en poco tiempo iba a convertir el mundo de las aplicaciones móviles en un negocio de miles de millones de dólares, llegaría de la mano de Apple con el cambio de milenio... aunque por una vía poco convencional: con música.

Según declaraciones de Steve Jobs, CEO de Apple, por aquel entonces el comité ejecutivo de la compañía no creía que las PDA o los *tablet* PC (ordenadores portátiles con pantalla táctil que permitía su uso sin teclado, como si fueran una tabla) fueran buenas opciones para conseguir un gran volumen de negocio para Apple. Jobs creía que el futuro del mercado estaba en los teléfonos móviles; que éstos iban a hacerse

dispositivos importantes para el acceso a la información de forma portátil, y que por tanto los teléfonos móviles debían tener una sincronización de software excelente. Por ello, en lugar de dedicarse a evolucionar su línea de PDA basada en el sistema Newton, los empleados de Apple pusieron todas sus energías en el reproductor de música iPod y la plataforma iTunes, un conjunto de *software* y tienda en línea que permitía a los usuarios del dispositivo la sincronización del mismo con su biblioteca de música así como la compra de nuevas canciones y discos de una manera fácil e intuitiva.

El éxito de la plataforma iTunes fue demoledor. No sólo rompió todos los esquemas de lo que hasta entonces había sido el negocio de las discográficas, liberalizando el mercado y cambiando para siempre el paradigma de la venta de música y otras obras con propiedad intelectual. Fue además la demostración más clara de que cuantos menos impedimentos se le pongan al usuario para que compre algo, más fácil es que lo compre; y la primera piedra de la estructura que faltaba en el mercado de las aplicaciones móviles para convertirlas en el negocio que son hoy en día: una plataforma de distribución que permita a los programadores y empresas publicar y dar visibilidad a sus aplicaciones reduciendo al máximo la inversión necesaria, y a los usuarios adquirir las aplicaciones de la manera más sencilla posible.

Figura 9. A la izquierda, Apple MessagePad (1992), la primera en ser llamada PDA. A la derecha, Apple iPhone (2007), el detonador de la explosión del mercado de aplicaciones móviles.

Ya en 2008, Apple creó dentro de la plataforma iTunes la App Store, una tienda en línea que permitía a los usuarios comprar y descargar directamente en el móvil o a través del software iTunes aplicaciones para el producto estrella de la compañía, el teléfono móvil iPhone. Con una política de aceptación muy estricta, los desarrolladores que consiguen cumplir todos los requisitos exigidos por Apple pueden publicar en esta tienda sus aplicaciones y obtener el 70% de la recaudación de su venta. Con estas condiciones, muchos desarrolladores adaptaron su trabajo y

sus aplicaciones a la plataforma iOS de Apple, basada en el lenguaje Objective-C y en entornos de desarrollo sólo disponibles para sistemas Mac Os, comercializados también por Apple.

El mercado de las aplicaciones móviles disponía por fin de todos los elementos necesarios para su eclosión final, y así inició su despegue imparable, arropado por dispositivos con la tecnología necesaria para posibilitar aplicaciones interesantes y por plataformas de distribución de las mismas a un coste prácticamente cero. No obstante, la mejor parte del pastel, la que más beneficios generaba, estaba reservada a aquellos que abrazaran las tecnologías de Apple, y además era una parte forzosamente pequeña ya que no muchos usuarios podían permitirse adquirir dispositivos como el iPhone, con un coste de varios cientos de dólares. Por este motivo, en 2008 el monopolio de Symbian aún no peligraba.

Afortunadamente, ese mismo año 2008 llegó la alternativa: Android. Mientras que el negocio de Apple estaba limitado a un único modelo y a una tecnología y lenguaje propietarios, Google se presentaba con un sistema operativo abierto, basado en Linux y Java y que podía ser incluido en cualquier dispositivo móvil independientemente del fabricante. Mediante numerosos acuerdos comerciales, Google consiguió que varios de los más importantes fabricantes de teléfonos móviles, que hasta ese momento estaban vendiendo sus terminales con sistemas operativos propietarios o con distintas versiones de sistemas Symbian o Microsoft, adoptaran el nuevo sistema operativo Android: un sistema operativo moderno, respaldado por una gran empresa como era Google y que les permitiría competir con el rey del mercado, el iPhone, ahorrándose de paso los costes de desarrollo del sistema.

Además, Android llegaba de la mano de una plataforma de distribución de aplicaciones, Android Market, similar a la de Apple pero unas políticas de aceptación mucho menos restrictivas. Con estas condiciones y con un entorno de desarrollo basado en el lenguaje Java y en herramientas disponibles gratuitamente tanto para sistemas Windows como para sistemas Linux/Unix, muchos desarrolladores y empresas con experiencia previa en Java se decantaron por Android como punto de entrada al mercado de las aplicaciones móviles.

Desde entonces, el ascenso de Android ha sido imparable. Si en 2008 tenía menos de un 1% de mercado, en 2011 ha conseguido desbancar a Symbian como sistema operativo más utilizado en *smartphones*, con un 53% del mercado.

Escenario actual

Hoy en día, el mercado de las aplicaciones móviles se reparte principalmente entre Apple con su sistema iOS y Google con su sistema Android.

En el caso de iOS, con tecnologías propietarias y sólo disponibles en sistemas del propio Apple, su fuerza está en una base de usuarios con un perfil económico medio-alto, que permite a los desarrolladores fijar unos precios más jugosos y que compensan de sobra la inversión inicial necesaria para entrar en la App Store. Así, actualmente más de la mitad de los beneficios generados mundialmente por la venta

de aplicaciones móviles corresponden a aplicaciones diseñadas para el sistema iOS.

En el caso de Android, su rápida expansión y disponibilidad en multitud de dispositivos de distintos fabricantes, así como unas tecnologías abiertas que permiten un ciclo de desarrollo de aplicaciones con un coste de inversión casi nulo, le han servido para copar el mercado de dispositivos y aprovechar esta posición de fuerza para atraer a más y mejores desarrolladores. Existe una gran comunidad de desarrolladores escribiendo aplicaciones para extender la funcionalidad de los dispositivos Android. Según datos ofrecidos por Google, la tienda de aplicaciones Android Market ha sobrepasado ya las 400.000 aplicaciones, sin tener en cuenta aplicaciones de otras tiendas no oficiales (como por ejemplo Samsung Apps, de Samsung, o la AppStore de Amazon). Multitud de aplicaciones están siendo portadas desde el sistema iOS al sistema Android, en busca de una base de usuarios más amplia, y es de esperar que Android se estabilice como sistema operativo dominante de aquí en adelante, igual que lo fue Symbian en su momento.

El sistema operativo Android

Android es un sistema operativo móvil, es decir, un sistema operativo diseñado para controlar dispositivos móviles como pueden ser un teléfono de tipo *smartphone*, una PDA o una *tablet* (ordenadores portátiles en los que la entrada de datos se realiza normalmente mediante una pantalla sensible al tacto), aunque existen versiones del sistema operativo utilizadas en otros dispositivos multimedia como por ejemplo televisores.

Figura 1. "Andy", el robot ideado como logotipo del sistema operativo Android.

Historia

La primera versión de Android fue desarrollada por la compañía Android Inc., una firma fundada en 2003 con el objetivo de crear *"(...) dispositivos móviles más inteligentes y que estén más atentos a las preferencias y la ubicación de su propietario"*. Aunque la evolución del trabajo que se llevaba a cabo en Android Inc. se mantuvo prácticamente en secreto, el gigante Google empezaba a mostrar interés en el negocio de los dispositivos y comunicaciones móviles y en 2005 adquirió la compañía, haciéndola subsidiaria de Google Inc. e incorporando a su plantilla a la mayoría de programadores y directivos originales.

A partir de ese momento, Google inició un proceso de búsqueda de acuerdos con diferentes operadores de comunicaciones y fabricantes de dispositivos móviles, presentándoles una plataforma abierta y actualizable y ofreciéndoles diferentes grados de participación en la definición y desarrollo de sus caracterísitcas. La compra por parte de Google de los derechos de varias patentes relacionadas con tecnologías móviles hizo saltar la liebre y los mercados comenzaron a especular con la inminente entrada de la compañía en el negocio de las comunicaciones móviles.

Finalmente, el 5 de noviembre de 2007, el sistema operativo Android fue presentado al mundo por la Open Handset Alliance, un recién creado consorcio de más de 70 empresas del mundo de las telecomunicaciones (incluyendo a Broadcom, HTC, Qualcomm, Intel, Motorola, T-Mobile, LG, Texas Instruments, Nvidia, Samsung... por nombrar algunas), liderado por Google y cuyo objetivo es el desarrollo de estándares abiertos para dispositivos móviles. Bajo esta premisa, Google liberó la mayor parte del código fuente del nuevo sistema operativo Android usando la licencia Apache, una licencia libre, gratuita y de código abierto.

La evolución del sistema operativo desde su presentación ha sido espectacular. Sucesivas versiones han ido incorporando funcionalidades como pantalla multi-táctil,

reconocimiento facial, control por voz, soporte nativo para VoIP (Voice over Internet Protocol), compatibilidad con las tecnologías web más utilizadas o novedosas como HTML5 y Adobe Flash, soporte de Near Field Communication, posibilidad de convertir el dispositivo en un punto de acceso WiFi, grabación de vídeo en 3D... así como ampliando el espectro de dispositivos soportados, pudiendo encontrar hoy en día Android en el corazón de teléfonos móviles, tablets, ordenadores portátiles y hasta televisores.

En la actualidad, los últimos datos indican que Android acapara más del 50% de cuota de mercado de *smartphones* a escala mundial, con un crecimiento en el último año de un 380% en número de unidades fabricadas; muy por delante de iOS, el sistema operativo del Apple iPhone que, con una cuota del 19%, ha relegado a su vez a Symbian OS a la tercera posición.

Diseño

El sistema operativo Android está compuesto de un núcleo basado en Linux, sobre el cual se ejecutan: por una parte, la máquina virtual Dalvik, basada en Java pero diseñada para ser más eficiente en dispositivos móviles; y por la otra, una serie de librerías o bibliotecas programadas en C y C++ que ofrecen acceso principalmente a las capacidades gráficas del dispositivo. Por encima de esta capa encontramos el marco de trabajo de aplicaciones, una colección de librerías Java básicas, adaptadas para su ejecución sobre Dalvik; y finalmente tenemos las aplicaciones Java, que son las que ofrecen la funcionalidad al usuario.

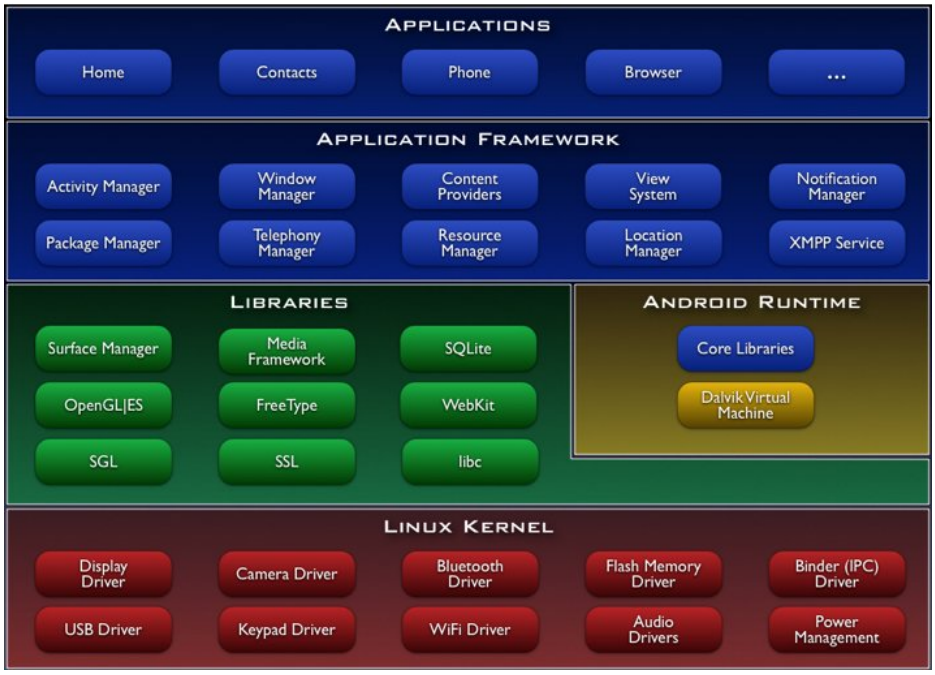

Figura 2. Las diferentes capas que conforman el sistema operativo Android.

A continuación presentamos en detalle cada uno de los componentes o capas que conforman el sistema operativo. Para cada uno se indica primero su denominación en inglés, en aras de una mejor identificación en el diagrama superior:

- ***Applications* (aplicaciones)**: las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros. Todas las aplicaciones están escritas en lenguaje de programación Java.
- ***Application framework* (marco de trabajo de aplicaciones)**: los desarrolladores tienen acceso completo a los mismos APIs del *framework* usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del *framework*). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario.
- ***Libraries* (bibliotecas)**: Android incluye un conjunto de bibliotecas de C/C++ usadas por varios componentes del sistema. Estas características se exponen a los desarrolladores a través del *framework* de aplicaciones de Android. Las bibliotecas escritas en lenguaje C/C++ incluyen un administrador de pantalla táctil (*surface manager*), un *framework* OpenCore (para el aprovechamiento de las capacidades multimedia), una base de datos relacional SQLite, una API gráfica OpenGL ES 2.0 3D, un motor de renderizado WebKit, un motor gráfico SGL, el protocolo de comunicación segura SSL y una biblioteca estándar de C, llamada "Bionic" y desarrollada por Google específicamente para Android a partir de la biblioteca estándar "libc" de BSD.
- ***Android runtime* (funcionalidad en tiempo de ejecución)**: Android incluye un set de bibliotecas base que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java. Cada aplicación Android corre su propio proceso, con su propia instancia de la máquina virtual Dalvik. Dalvik ha sido escrito de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecuta archivos en el formato Dalvik Executable (.dex), el cual está optimizado para memoria mínima. La Máquina Virtual está basada en registros y corre clases compiladas por el compilador de Java que han sido transformadas al formato.dex por la herramienta incluida "dx".
- ***Linux kernel* (núcleo Linux)**: Android dispone de un núcleo basado en Linux para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores, y también actúa como una capa de abstracción entre el hardware y el resto de la pila de software. La versión para Android se encuentra fuera del ciclo normal de desarrollo del kernel Linux, y no incluye funcionalidades como por ejemplo el sistema X Window o soporte para el conjunto completo de librerías GNU, lo cual complica bastante la adaptación de aplicaciones y bibliotecas Linux para

su utilización en Android. De igual forma, algunas mejoras llevadas a cabo por Google sobre el núcleo Linux original han sido rechazadas por los responsables de desarrollo de Linux, argumentando que la compañía no tenía intención de dar el soporte necesario a sus propios cambios; no obstante ya se han dado los pasos necesarios para asegurar que todas estas mejoras y controladores se incluyan en las próximas versiones del núcleo oficial.

Estructura básica de una aplicación Android

Las aplicaciones Android están escritas en lenguaje Java. Aunque ya hemos visto que la máquina virtual Dalvik no es una máquina virtual Java, Android toma ventaja de la utilización de un lenguaje de programación consolidado y de libre acceso como es Java para facilitar a los programadores el desarrollo de aplicaciones para su sistema. Así, el proceso de desarrollo y pruebas se puede llevar a cabo usando cualquier entorno de programación Java, incluso la simulación mediante dispositivos Android virtuales, dejando que sea el sistema operativo Android el que en tiempo de ejecución traduzca el código Java a código Dalvik. Si queremos distribuir la aplicación una vez completado el desarrollo de la misma, el SDK de Android compila todo el código Java y, junto con los ficheros de datos y recursos asociados (imágenes, etc.), crea un "paquete Android": un archivo con extensión .apk que representa a una única aplicación y que es el archivo utilizado por los dispositivos Android para instalar nuevas aplicaciones.

Estas aplicaciones, una vez instaladas, se ejecutarán como un nuevo proceso en el sistema Linux subyacente, utilizando su propio identificador único de usuario y su propia instancia de la máquina virtual Dalvik. Cuando la aplicación deja de utilizarse o cuando el sistema necesita memoria para otras tareas, el sistema operativo Android finaliza la ejecución del proceso, liberando los recursos. De esta forma, Android crea un entorno bastante seguro en el que cada aplicación, por defecto, sólo puede acceder a los componentes estrictamente necesarios para llevar a cabo su tarea, ya que cada proceso está aislado del resto y no puede acceder a partes del sistema para las cuales no tiene permisos.

Como es normal, Android dispone de diferentes maneras de que dos aplicaciones puedan compartir recursos. El más habitual es el sistema de permisos, mediante el cual una aplicación indica qué funcionalidades del sistema necesitará utilizar; la responsabilidad de permitir el acceso recae en el usuario, quien puede aceptar o denegar dichos permisos durante la instalación de la aplicación.

Componentes de una aplicación

Según las tareas que necesite llevar a cabo, una aplicación Android puede estar formada por uno o varios de los siguientes componentes:

- **Actividades** (implementadas como subclases de `Activity`)

  Una actividad representa una pantalla de la aplicación con la que el usuario interactúa. Por ejemplo, una aplicación que reproduzca música podría tener

una actividad para navegar por la colección de música del usuario, otra actividad para editar la lista de reproducción actual, y otra actividad para controlar la canción que se está escuchando actualmente. Aunque una aplicación puede tener múltiples actividades, que juntas ofrecen la funcionalidad completa de la aplicación, las actividades son elementos independientes. Esto implica que otra aplicación distinta podría invocar cualquiera de estas actividades para utilizar su funcionalidad, si tiene los permisos adecuados. Por ejemplo, una aplicación de selección de tonos de llamada podría invocar la actividad que muestra la colección de música del usuario para permitirle elegir una canción como tono personal. Otro ejemplo que podemos ver a diario es la utilización de la actividad "Cámara" (incluida por defecto en Android) cada vez que una aplicación quiere permitir al usuario tomar una fotografía, ya sea para enviarla a un amigo, para fijarla como fondo de pantalla, para subirla a redes sociales, para dibujar sobre ella...

- **Servicios** (implementados como subclases de `Service`)

  Un servicio no tiene interfaz de usuario, ya que se ejecuta en segundo plano para llevar a cabo operaciones de mucha duración y funcionalidades que, en general, no requieren de intervención del mismo. Siguiendo con nuestro ejemplo de la aplicación musical, ésta podría implementar un servicio que continúe reproduciendo canciones en segundo plano mientras el usuario lleva a cabo otras tareas, como comprobar su correo o visualizar una presentación de fotografías. La descarga de contenidos desde Internet también se suele llevar a cabo mediante un servicio en segundo plano, que notifica la finalización de la misma a la aplicación original.

- **Proveedores de contenido** (implementados como subclases de `ContentProvider`)

  Un proveedor de contenidos es un componente que controla un conjunto de datos de una aplicación. El desarrollador puede almacenar estos datos en cualquier ubicación a la que su aplicación pueda acceder (sistema de archivos, base de datos SQLite, servidor web) y luego crear un proveedor de contenidos que permita a otras aplicaciones consultar o incluso modificar estos datos de manera controlada. El reproductor de música podría incluir un proveedor de contenidos que permita a otras aplicaciones consultar el catálogo de música del usuario y añadir nuevas entradas al mismo. Android ofrece a todas las aplicaciones con permiso para ello acceso a la lista de Contactos, también mediante un proveedor de contenidos.

- **Receptores de notificaciones** (implementados como subclases de `BroadcastReceiver`)

  Un receptor de notificaciones responde a notificaciones lanzadas "en abierto" para todo el sistema. Muchas notificaciones las lanza el propio sistema, por ejemplo avisos de batería baja, de que la pantalla se ha apagado o de que hay una llamada entrante. Las aplicaciones también pueden lanzar notificaciones,

por ejemplo para informar al resto del sistema de que se ha descargado algún tipo de información y que ya está disponible. Aunque los receptores de notificaciones no tienen interfaz de usuario, sí que pueden informar a éste de los eventos detectados, mediante un icono en la barra de notificaciones. En nuestro ejemplo, el reproductor de música podría tener un receptor de notificaciones para detectar las llamadas entrantes y detener la reproducción en curso o pasar el reproductor a segundo plano.

En Android, cualquier aplicación puede ejecutar cualquier componente de otra aplicación distinta. Es una característica única del sistema operativo Android que facilita muchísimo el desarrollo de nuevas aplicaciones ya que permite reutilizar, para funcionalidades comunes, componentes ya existentes en otras aplicaciones, pudiendo así centrar los esfuerzos de desarrollo en las funcionalidades y componentes nuevos que generen valor añadido. Por ejemplo, podríamos crear una aplicación de tipo videojuego que permita asignarle a cada jugador una fotografía. En otro sistema operativo sería necesario codificar toda la lógica de acceso a la cámara del dispositivo y la interfaz de usuario para ello, o al menos incluir en nuestra aplicación las llamadas a la API y las librerías necesarias; en Android podemos simplemente invocar la actividad Cámara que, una vez finalizada su ejecución, nos retornará la imagen capturada.

Para conseguir esto, dado que por seguridad cada aplicación se ejecuta aislada de las demás y no tiene acceso directo al resto de aplicaciones, lo que tenemos que hacer es indicarle al sistema operativo que tenemos *intención* de lanzar una actividad o componente. Esta solicitud se efectúa mediante la creación de un objeto `Intent` con el identificador del componente o servicio. El sistema entonces comprueba los permisos, ejecuta (o reactiva) el proceso asociado a la aplicación propietaria del componente, e instancia las clases necesarias para ofrecer la funcionalidad solicitada. La lógica de estas clases se ejecutará en el proceso de la aplicación propietaria, no de la aplicación usuaria, con lo cual el componente sigue estando restringido por los permisos que tenía originalmente.

Manifiesto de la aplicación

Para que Android pueda ejecutar un componente cualquiera de una aplicación, la aplicación debe primero informar al sistema de cuáles son sus componentes mediante el fichero AndroidManifest.xml, conocido como "manifiesto" de la aplicación. En este fichero, cada aplicación proporciona entre otras cosas los siguientes datos:

- **Componentes de la aplicación**: actividades, servicios y proveedores de contenido deben ser declarados obligatoriamente en este fichero, mediante el nombre canónico de su clase principal. Este será el identificador utilizado en los objetos Intent por el resto de aplicaciones para solicitar al sistema la ejecución de un componente. Los receptores de notificaciones son los únicos componentes que además de en el manifiesto pueden "registrarse" en el sistema posteriormente, en tiempo de ejecución.
- **Capacidades de los componentes**: aunque una aplicación puede ejecutar un

componente externo invocándolo directamente por su nombre canónico, existe un método más potente, basado en acciones. Utilizando acciones, la aplicación que crea el objeto Intent sólo tiene que indicar qué acción quiere llevar a cabo, dejando que sea el sistema el que busque componentes que puedan llevarla a cabo; por ejemplo, una aplicación podría indicar que tiene la intención de ejecutar la acción ACTION_SEND (enviar un mensaje), el sistema relacionaría la acción con la actividad "enviar" del programa de mensajería y lanzaría dicha actividad al ser invocada esta acción. En el manifiesto de la aplicación, cada componente puede indicar opcionalmente una lista de acciones que está preparado para llevar a cabo, de manera que el sistema lo tendría en cuenta a la hora de ejecutar cualquiera de esas acciones.

- **Permisos necesarios**: cada aplicación debe informar en su manifiesto de todos los permisos que necesita para su funcionamiento. El usuario no puede aceptar o denegar permisos individuales: o bien los acepta todos e instala la aplicación, o los deniega todos y cancela la instalación.
- **Librerías y API externas utilizadas**: librerías externas a la propia API del sistema Android, que la aplicación utiliza para funcionar; por ejemplo, las librerías de Google Maps.
- **Requisitos mínimos**: en el manifiesto se indican tanto el nivel mínimo de la API de Android obligatorio para el funcionamiento de la aplicación, como características específicas de hardware y software sin las cuales la aplicación no pueda desarrollar sus capacidades. Por ejemplo, si nuestra aplicación utiliza las capacidades *multitouch* de Android (sensibilidad a múltiples puntos de contacto), tendremos que especificar en el manifiesto que nuestra aplicación requiere una pantalla capaz preparada para *multitouch*, y al menos un nivel 5 de API (primer nivel en el que se introdujeron los eventos relacionados).

El manifiesto de una aplicación se encuentra en el directorio raíz del paquete .apk y, aunque el sistema operativo no utiliza toda la información que contiene, otras aplicaciones como Android Market sí que utilizan esta información para saber si una aplicación es compatible con el dispositivo que está accediendo al servicio.

Recursos de la aplicación

Además de clases Java y el manifiesto, una aplicación Android se acompaña normalmente de una serie de recursos separados del código fuente, como pueden ser imágenes y sonidos, ficheros XML para las visuales y cualquier otro recurso relacionado con la presentación visual o la configuración de la aplicación. Estos recursos se organizan en directorios dentro del paquete .apk como sigue (se muestran sólo los más habituales):

- **res**: directorio raíz de los recursos.
    - **drawable**: este directorio contiene imágenes en formato PNG, JPG o GIF, que después se cargarán en la aplicación como objetos de tipo `Drawable`.

- **layout**: directorio para almacenar las visuales XML de la aplicación, es decir, ficheros XML que contienen la definición de las interfaces de usuario.
- **menu**: contiene ficheros XML que definen los menús utilizados en la aplicación.
- **values**: agrupa ficheros XML que definen múltiples recursos de tipos sencillos, como cadenas de texto predefinidas, valores enteros constantes, colores y estilos. Para definir cadenas se suele utilizar el fichero `strings.xml`, para definir estilos el fichero `styles.xml`, etc.

- **assets**: directorio para almacenar ficheros que la aplicación accederá de manera clásica, mediante la ruta y el nombre, sin asignarle un identificador único ni transformarlo antes a un objeto equivalente. Por ejemplo, un grupo de ficheros HTML que se mostrarán en un navegador web, deben poder accederse mediante su ruta en el sistema de archivos para que los hiperenlaces funcionen.

Para cada uno de estos recursos (excepto los contenidos del directorio `assets`), el SDK de Android genera un identificador único y lo almacena como una constante dentro de la clase `R.java`. Así, una imagen situada en `res/drawable/thumb1.png` podrá referenciarse mediante la variable `R.drawable.thumb1`, y `R.layout.principal` apuntaría a una visual XML guardada en `res/layout/principal.xml`.

Además, el desarrollador puede añadir sufijos a cualquiera de los directorios para definir recursos que sólo estarán disponibles cuando se den las condiciones definidas por ese sufijo. Un uso muy habitual es definir directorios `drawable-ldpi` y `drawable-hdpi` para almacenar recursos que sólo deben utilizarse en casos de que el dispositivo tenga una resolución de pantalla baja (ldpi) o alta (hdpi). Otro uso es disponer de versiones "localizadas" de las distintas cadenas de texto utilizadas en la aplicación, creando directorios como `res/values-en` y `res/values-de` para guardar la traducción al inglés y al alemán de los textos de nuestra aplicación.

Con esto finalizamos la introducción al sistema operativo Android y sus aplicaciones. Para conocerlo en mayor profundidad, se puede acudir a la bibliografía, aunque es más edificante construir nuestra propia aplicación Android y aprender sobre la marcha.

Patrones de diseño

Uno de los objetivos de este proyecto, es mostrar cómo se utilizarían en una aplicación "real" algunos de los patrones de diseño *software* vistos a lo largo de la carrera. De todos los explicados en las distintas asignaturas relacionadas con la programación y la ingeniería del software, hay dos que encajan bastante bien con diferentes aspectos de Android y de nuestra aplicación: son el patrón *Singleton* y el patrón Modelo-Vista-Controlador (MVC).

El patrón Singleton

El patrón de diseño *Singleton* (de *single*, único) está diseñado para restringir la creación de objetos pertenecientes a una clase a un único objeto o instancia. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación. Es este último caso el que justifica la utilización de dicho patrón en nuestra aplicación.

Como veremos más adelante, una de las clases Java de la aplicación se encarga precisamente de controlar la generación e inicialización de todos los datos de los edificios: nombres, descripciones, coordenadas, puntos de interés, etc.
Es, por tanto, un escenario típico en el que la utilización del patrón *Singleton* nos proporciona:

- por una parte, un mecanismo para evitar incoherencias en los datos obtenidos, al designar un único punto de acceso a dichos datos que es global para todas las actividades;
- por otra parte, un importante ahorro en la memoria total utilizada por la aplicación, dado que con este patrón se evita la instanciación de múltiples objetos que contendrían los mismos datos, lo cual supondría una duplicidad de información innecesaria.

El patrón *Singleton* se aplica implementando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe ninguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado). Una implementación típica sería la siguiente:

```
public class Singleton {

	// Instancia única de la clase
	private static Singleton INSTANCE;

	private Singleton() {
		// Privado para que no se pueda llamar directamente
	}

	public static Singleton getInstance() {
		if(INSTANCE == null) {
			// Si es la primera invocación, inicializamos
la instancia única
			INSTANCE = new Singleton();
		}
		// Devolvemos la misma instancia siempre
		return INSTANCE;
	}

}
```

Esta implementación, además, favorece que la creación de la instancia de la clase se postergue hasta que haga falta *(lazy inicialization*, inicialización "perezosa"), consiguiendo así ahorrar en la pila el espacio destinado al objeto mientras éste no sea necesario. Sin embargo, la instrumentación del patrón así definido puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. En el ejemplo anterior, si dos hilos llaman al método `getInstance()` al mismo tiempo cuando `INSTANCE` todavía es nulo, ambos podrían entrar en la lógica de llamada al constructor y obtener instancias distintas de la clase.

La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón. En el ejemplo anterior, el método `getInstance()` debería declararse como `synchronized` para asegurar que sólo uno de los hilos llega a crear la instancia, estando la misma ya creada cuando al otro hilo se le permite el acceso al método.
No obstante, cuando no hay un interés especial en que la inicialización sea de tipo *lazy*, existe una opción más sencilla, basada en la implementación del patrón ideada por el profesor William "Bill" Pugh durante sus estudios sobre el modelo de memoria de Java:

```
public class Singleton {

	// Instancia única de la clase
	private static Singleton INSTANCE = new
Singleton();

	private Singleton() {
		// Privado para que no se pueda llamar
directamente
	}

	public static Singleton getInstance() {
		// Devolvemos la misma instancia siempre
		return INSTANCE;
	}

}
```

Esta variante es segura en entornos multi-hilo gracias a que se aprovecha de la lógica interna de inicialización de clases garantizada por la especificación de Java, la cual es compatible con la implementación nativa de Android, Dalvik; consiguiendo así un objeto *singleton* sin necesidad de recurrir a construcciones específicas de sincronización de hilos como *synchronize.* Por su sencillez ha sido la elegida a la hora de implementarla en el proyecto actual.

El patrón MVC (Model-View-Controller)

Modelo Vista Controlador, o MVC, es un patrón de arquitectura *software* que separa una aplicación en tres componentes o partes diferenciadas: el Modelo, que representa los datos de una aplicación; la Vista, normalmente equivalente a la interfaz de usuario; y el Controlador, la lógica que procesa las acciones y eventos generados durante el tiempo de vida de la aplicación; de manera que entre estos componentes existe lo que se llama "bajo acoplamiento", es decir, que la definición y detalles de cada uno de ellos (y en el caso de la Vista, hasta su propia existencia) son más o menos desconocidos para los otros dos. Este patrón fue descrito por primera vez en 1979 por el científico computacional Trygve Reenskaug, durante su trabajo en el lenguaje de programación Smalltalk en el centro de investigación de la compañía Xerox.
A continuación se presenta una explicación algo más detallada de cada componente:

- **Modelo**: representa los datos con los que trabaja la aplicación y, en muchas ocasiones, también incluye la lógica de negocio utilizada para manejarlos. Responde a peticiones de consulta sobre su estado y de actualización del mismo, hechas por la Vista y el Controlador respectivamente, pero no suele tener conocimiento de la existencia de estos otros componentes como tales. Al ser el Modelo el componente asociado a la definición "formal" del sistema, una vez implementada es la parte que menos cambia en una aplicación que siga el

patrón MVC.

- **Vista**: recoge la información del Modelo y la presenta al usuario en una forma tal que pueda ser interpretada y manejada por el mismo, casi siempre a través de una interfaz gráfica; engloba también toda la lógica de interactuación con los dispositivos físicos que conforman dicha interfaz de usuario (pantallas, teclados, superficies sensibles al tacto), incluyendo la generación de los eventos necesarios, que serán recibidos por el Controlador. Gracias al bajo acoplamiento con el Modelo, una misma aplicación puede tener diferentes Vistas intercambiables entre sí para presentar la información de diferentes maneras.
- **Controlador**: recibe los eventos generados por el usuario a través de la Vista, los procesa y genera las respuestas adecuadas en el Modelo. En ocasiones la acción del usuario no provoca una actuación sobre el Modelo, sino sobre la propia Vista; en tal caso el Controlador se encarga también de iniciar las rutinas necesarias para que la Vista refleje estos cambios.

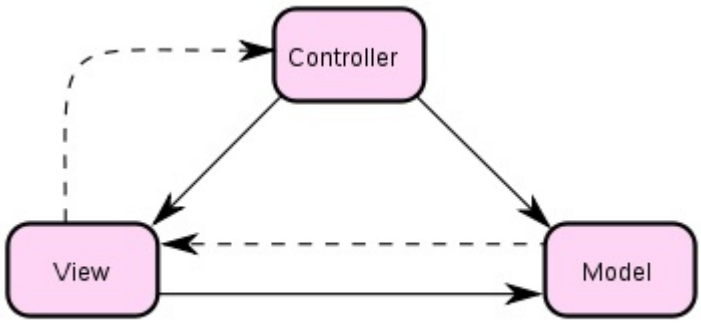

Figura 1. Diagrama ideal de relación entre los componentes del patrón MVC. La Vista tiene conocimiento del Modelo, y el Controlador de ambos, pero no a la inversa.

Existen varias implementaciones del patrón MVC, aunque en todas ellas el flujo general de control viene a ser el siguiente:

1. El usuario lleva a cabo algún tipo de acción sobre la interfaz de usuario; esto es, sobre la Vista. Por ejemplo, pulsar un botón.
2. El Controlador, normalmente tras registrarse como receptor, recibe de la Vista el evento generado por la acción del usuario y lo gestiona.
3. En caso necesario (si no es una simple consulta), el Controlador accede al Modelo para actualizar su estado, por ejemplo eliminando un registro; o simplemente notifica al Modelo la operación a realizar, en el caso de que los objetos del Modelo incluyan su propia lógica de negocio interna.
4. El Modelo actualiza su estado.
5. La Vista actualiza la interfaz de usuario para reflejar los cambios llevados a cabo en el Modelo.
6. La aplicación queda a la espera de la siguiente acción del usuario.

Aproximaciones al patrón MVC

En las implementaciones de un patrón MVC puro, Controlador y Vista deben mantener un alto grado de desacoplamiento. El Controlador no interviene en el proceso de actualización de la Vista, dejando que sea ella misma la que detecte los cambios en el Modelo, mediante eventos o por consulta directa. De igual forma, la Vista no incluye la lógica para gestionar los eventos del usuario; éstos son recibidos directamente por el Controlador, que es quien decide la actuación que se debe llevar a cabo en base a cada evento.

No obstante, mantener este grado de desacoplamiento no siempre es fácil y, en ocasiones, tampoco es productivo puesto que fija a los desarrolladores de cada componente unos límites a veces artificiales. Como respuesta a estas situaciones, basándose en el patrón MVC original se han ideado múltiples patrones derivados que trasladan hacia un lado (Vista) u otro (Controlador) el peso de la lógica principal de la aplicación, según convenga al escenario y tipo de aplicación concretos.

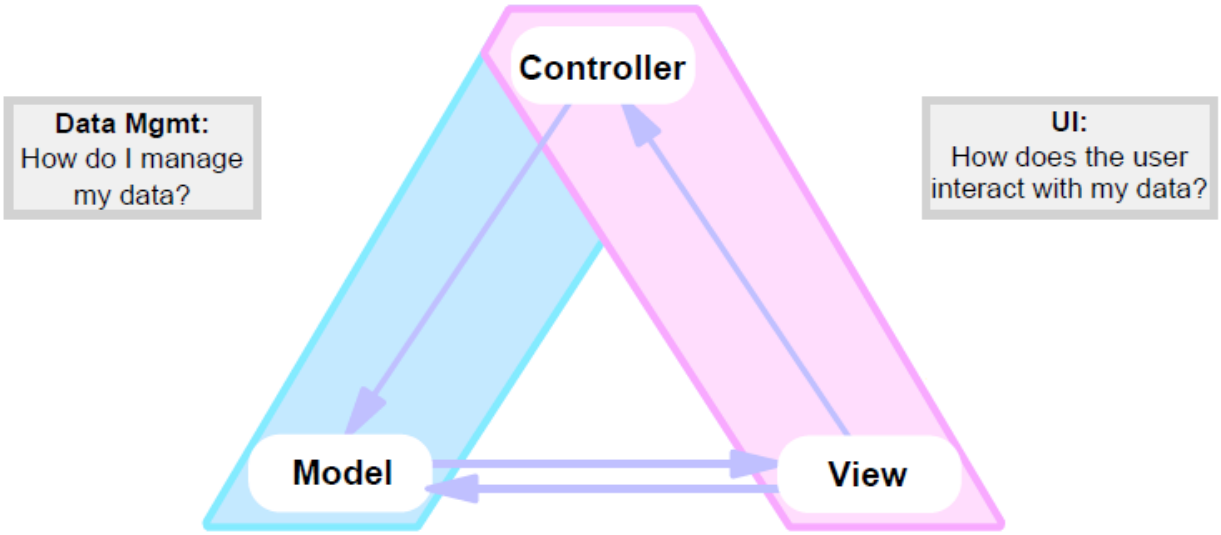

Figura 2. Ilustración de la distinción que el patrón MVP (Modelo-Vista-Presentador) hace entre Modelo y Presentación.

Por ejemplo, hay escenarios en los que interesa centralizar en la Vista todas las acciones relacionadas con la interfaz de usuario, siendo la propia Vista la receptora de los eventos generados por el usuario, no el Controlador; mientras que el Controlador se encarga del tratamiento de los datos y de adaptar el formato de los mismos de manera que puedan ser procesados fácilmente por la Vista, pero sin conocer directamente las acciones que el usuario genera, sólo aquellas que la Vista delega en él.

Esta variante del patrón MVC es conocida como "Modelo-Vista-Presentador" o MVP (*Model-View-Presenter*), y fue derivada directamente de la implementación que incluía Smalltalk de MVC. Su objetivo es la simplificación del modelo en dos partes correspondientes a las dos cuestiones que se plantean durante el desarrollo de una aplicación: por un lado, cómo maneja el programa los datos, de lo cual se encarga la lógica de comunicación entre el Controlador y el Modelo; y por el otro, cómo interactúa el usuario con dichos datos, cuestión que resuelve la Presentación, es decir la combinación de la Vista y el Presentador (o Controlador).

En otros casos, la Vista no tiene –o no conviene que tenga– acceso directo al Modelo. Una solución es que sea el Controlador el que recoja los datos del Modelo, asumiendo parte de las funciones de la Vista en el sentido de que pasa a ser el que envía y recibe los datos, y quedando la Vista únicamente como una capa gráfica controlada totalmente por el Controlador. En este escenario, la premisa de bajo acoplamiento entre ambos componentes del patrón no se cumple, y el Controlador se convierte en una especie de segunda Vista que puede acceder directamente al Modelo, o en un segundo Modelo conceptualizado de una forma más cercana a la presentación que se hace del mismo al usuario. Es por ello que en la literatura especializada, este tipo de Controlador es referido como "Modelo de la Vista" y el patrón resultante es llamado MVVM (*Model-View-ViewModel*).

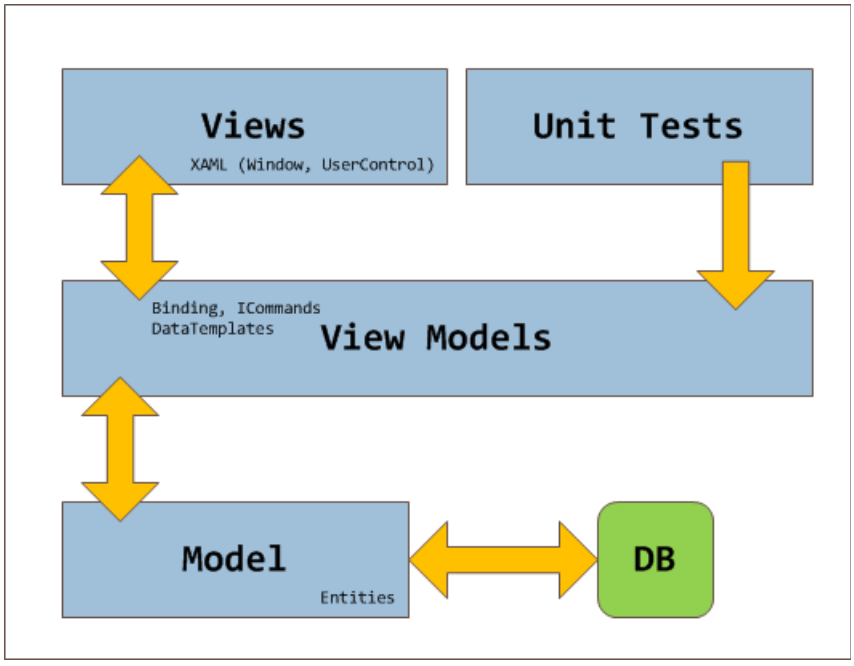

Figura 3. Relación entre los distintos componentes del patrón MVVM: Modelo, Modelo de la Vista, y Vista.

Estas variantes del patrón MVC se adaptan mejor a la arquitectura general de una aplicación Android que un patrón MVC puro, como veremos más adelante, debido a que el alto acoplamiento que existe entre las clases `Activity` y la clase o clases `View`

asociadas a las mismas propicia el trasvase de competencias entre el Controlador (la clase `Activity` y auxiliares) y la Vista (las clases `View` y sus visuales definidas en XML).

Esto no quiere decir que los conceptos relativos al patrón MVC no sean aplicables a nuestro programa. Aunque no pueda considerarse un patrón MVC puro, muy pocas aplicaciones del mundo real consiguen una implementación exacta del patrón MVC debido a la dificultad de lograr un desacoplamiento total entre los tres componentes, y la variante utilizada a lo largo de este proyecto representa un didáctico ejemplo de aproximación al modelo ideal MVC en una aplicación real.

Entorno de desarrollo

Aunque a veces no se le da la importancia que merece, la elección de un buen entorno de desarrollo es uno de los puntos básicos a la hora de enfocar un nuevo proyecto. Comenzar a trabajar con herramientas que no son las óptimas puede causar inconvenientes que, dependiendo de su naturaleza, generarán retrasos o frustración que se irán acumulando conforme avance el proyecto y afectarán al desarrollo del mismo. Un entorno de trabajo óptimo, en cambio, facilitará el trabajo diario, automatizando tareas rutinarias y favoreciendo que la mayor parte del esfuerzo durante la etapa de desarrollo se dedique a construir la aplicación en sí misma.

A continuación se presenta una descripción de las herramientas que conforman el entorno de trabajo utilizado durante el proyecto, resaltando los puntos que han llevado a su elección sobre otras similares.

Android SDK

Independientemente del resto de herramientas con las que elijamos trabajar, lo primero que necesitamos para desarrollar aplicaciones para Android es el Android SDK (*Software Development Kit*, kit de desarrollo de software): el conjunto de herramientas básicas que permiten compilar y depurar aplicaciones escritas para el sistema operativo Android, así como empaquetar y firmar las aplicaciones para su posterior distribución (por ejemplo, a través de Android Market).

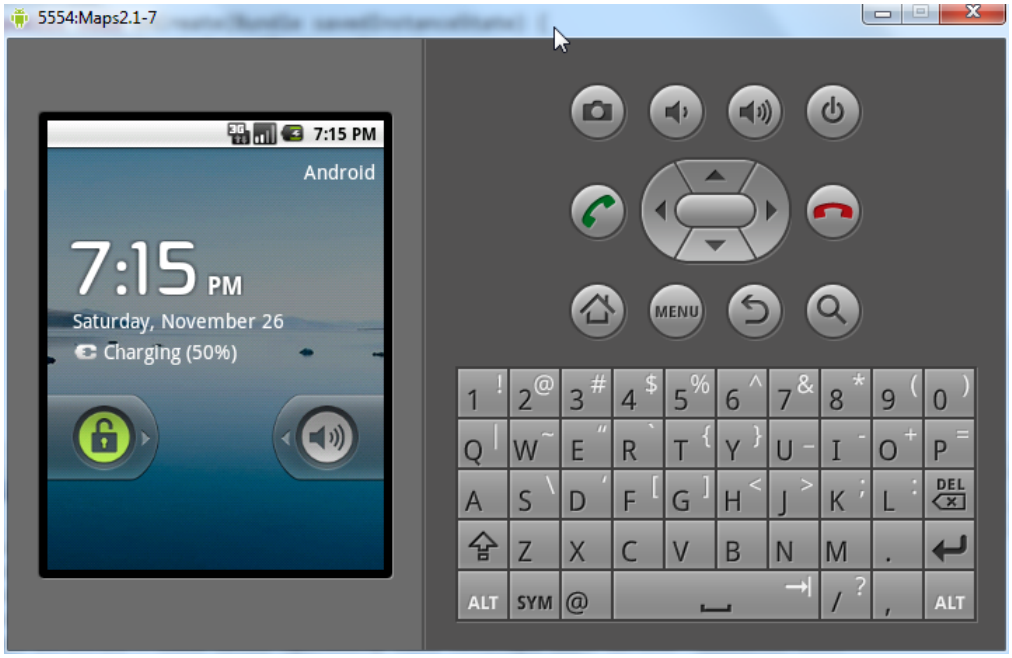

Figura 1. Aspecto del emulador de dispositivo Android incluido con el SDK.

Algunas de las herramientas más importantes son:

- Librerías de la API de Android, correspondientes a cada una de las versiones de Android que se quieran utilizar.
- Herramientas de compilación, *profiling*, depuración, empaquetado y análogas, en su versión de línea de comandos.
- Emulador de dispositivo Android, que permite ejecutar las aplicaciones en un dispositivo virtual simulando la pantalla, teclado y controles de un terminal Android.
- Controlador USB y herramientas para poder ejecutar y depurar las aplicaciones utilizando un dispositivo real, en vez del emulador.
- DDMS (Dalvik Debug Monitor Server) que permite controlar aspectos del dispositivo Android más allá de lo que permite el propio dispositivo: pila de procesos e hilos, simulación de recepción de datos GPS arbitrarios, control de la latencia de la conexión de datos, simulación de llamadas entrantes, acceso a los logs del sistema y varias funcionalidades más.
- Documentación y aplicaciones sencillas de ejemplo que muestran distintos aspectos de la API.
- Librerías externas adicionales, como por ejemplo la API de Google Maps.
- Herramientas con interfaz gráfica para la gestión de los distintos componentes, permitiendo actualizarlos y descargar nuevas versiones de las herramientas.

Las herramientas anteriores son las que nos permitirán tomar el código Java, compilarlo, ejecutarlo en un dispositivo (virtual o no), depurarlo y finalmente crear los binarios de nuestra aplicación Android. Nos falta sin embargo un editor que nos permita crear el código Java en primer lugar.

Eclipse

Eclipse es una plataforma de desarrollo abierta y libre, extensible mediante módulos y *plug-ins*. Está orientada inicialmente a la creación de aplicaciones Java mediante sus funcionalidades como entorno de desarrollo integrado (IDE, por sus siglas en inglés); sin embargo, su código abierto, sus posibilidades de extensión y la posibilidad de crear aplicaciones completas utilizando la propia plataforma Eclipse como base, han hecho que se cree una gran comunidad a su alrededor que proporciona, además de compatibilidad con múltiples lenguajes adicionales (desde Ada hasta Python pasando por C++, Scheme, Perl o COBOL), módulos de extensión y herramientas para cualquier tarea que se nos presente a lo largo del ciclo de vida de nuestra aplicación. ¿Necesitas control de versiones? Eclipse permite de forma nativa trabajar con repositorios CVS y, mediante complementos, con otros sistemas más avanzados como Subversion, Mercurial o git. ¿Estás desarrollando una aplicación web con JSP y servlets? Eclipse descarga automáticamente *plug-ins* para los contenedores más conocidos (Tomcat, WebLogic, etc.) y gestiona tanto el arranque y parada del servidor como todo el proceso de empaquetado, publicación y despliegue de la aplicación web. ¿Depuración de código Java? Eclipse ofrece múltiples opciones de ejecución en modo depuración,

puntos de interrupción, seguimiento de variables e incluso sustitución de código "en caliente" (si la máquina virtual lo permite). ¿Diagramas UML? El sitio web de Eclipse tiene secciones enteras dedicadas a herramientas y módulos enfocados al "desarrollo por modelos".

Eclipse surgió como un proyecto de la división canadiense de IBM para sustituir su línea de productos VisualAge, desarrollados en SmallTalk, por una plataforma más moderna basada en Java. En 2001 se decidió liberar el código del proyecto, creándose un consorcio de más de 50 empresas que apoyó el desarrollo abierto de la plataforma. La Fundación Eclipse se creó a partir del consorcio original y actualmente está compuesta por más de 180 miembros, como por ejemplo: Adobe, AMD, Borland, Cisco, Google, HP, IBM, Intel, Motorola, Nokia, Novell, Oracle, SAP o Siemens, por citar algunos de los más conocidos.

Actualmente, Eclipse es una plataforma ampliamente usada (quizá la que más) tanto por desarrolladores independientes como en entornos empresariales, dado que la mayoría de herramientas, módulos y *plug-ins* existentes para la misma son libres y gratuitos y representa, por tanto, la mejor opción calidad-precio a la hora de elegir un entorno de desarrollo potente y fiable, teniendo un coste prácticamente nulo.

Pero, por encima de todo, Eclipse es el IDE recomendado por los propios creadores de Android, y de hecho es el único que dispone oficialmente de un plugin, ADT, que facilita enormemente todo el proceso de desarrollo, desde la gestión de librerías hasta la depuración de la aplicación y la creación de los paquetes .apk que contienen todos los recursos. Esto quiere decir que si elegimos otro IDE distinto (como podría ser NetBeans, por ejemplo) nos veremos obligados a ejecutar manualmente tareas que, como veremos ahora, en Eclipse se verán completamente integradas en el flujo de trabajo.

ADT para Eclipse

ADT (*Android Development Tools*, herramientas de desarrollo Android) es un *plug-in* diseñado específicamente para Eclipse por los creadores de Android. Este *plug-in* incorpora a Eclipse menús y opciones que facilitan sobremanera tareas habituales en el desarrollo de aplicaciones para Android: compilación automática de las clases, instalación de la aplicación en los dispositivos utilizados durante el desarrollo, interfaz gráfica para previsualizar las visuales XML, conexión con los dispositivos en modo depuración... Además de estar absolutamente integrado con la herramienta DDMS.

Mediante este *plug-in*, Eclipse se convierte en el entorno ideal para el desarrollo de aplicaciones en Android, y junto con las herramientas anteriores completa el entorno de trabajo seleccionado para el proyecto.

Ejemplo de desarrollo

A continuación vamos a ver un ejemplo muy resumido de cómo construir paso a paso, desde cero y utilizando las herramientas arriba mencionadas, una aplicación Android que muestre por pantalla el mensaje "¡Mi primer proyecto Android!".

Este ejemplo está basado en el tutorial *Hello, World* que se encuentra en la página web oficial de Android para desarrolladores, referenciada en la bibliografía. El detalle de cada uno de los pasos se puede consultar en dicho tutorial, ya sea de manera *online*, o en la versión *offline* que se descarga junto con la documentación del SDK de Android.
Se da por hecho, así mismo, que tenemos instaladas y configuradas todas las herramientas anteriores, y que ya hemos descargado al menos una versión de Android sobre la cual desarrollar nuestro ejemplo. Si no es así, se recomienda consultar la bibliografía para ampliar la información acerca de cómo hacerlo.

Los pasos a seguir para la creación de nuestro primer proyecto son los siguientes:

1. **Creación de un dispositivo Android virtual**. Para probar nuestra aplicación Android necesitaremos un dispositivo en el que ejecutarla, lo más sencillo es crear uno virtual. Para ello sólo tenemos que acceder desde Eclipse al *Android SDK and AVD Manager*, crear un nuevo dispositivo en el apartado *Virtual Devices* y seleccionar qué versión de Android queremos que utilice, de entre todas las que hayamos descargado.

2. **Creación del proyecto Android en Eclipse**. El siguiente paso es crear el proyecto sobre el cual trabajaremos desde Eclipse. Al tener instalado el *plug-in* ADT, es tan sencillo como seleccionar Android Project como tipo del nuevo proyecto y rellenar los campos necesarios: nombre del proyecto, versión de Android utilizada, etc. Por coherencia con el resto del ejemplo, llamaremos al proyecto "Hello Android". Una vez que aceptemos la creación del nuevo proyecto, ADT generará automáticamente las clases Java, visuales XML y recursos necesarios para definir una nueva actividad; nosotros sólo tenemos que modificar la visual y el controlador para adaptarlos a nuestras necesidades.

3. **Construir la visual XML**. Un proyecto Android puede tener tantas visuales XML como nosotros queramos, todas ellas alojadas en el directorio `res/layout` del proyecto; la visual por defecto se llama `main.xml` y se crea al mismo tiempo que el proyecto. Podemos desecharla si queremos, pero en nuestro caso es más sencillo editarla para dejarla de la siguiente manera:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/textview"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:text="¡Mi primer proyecto Android!"/>
```

Con esto simplemente definimos una visual cuyo único contenido será un elemento `TextView` (una etiqueta), que llenará todo el especio disponible en la pantalla y que mostrará el texto indicado.

4. **Adaptar la lógica Java del controlador**. En lo que a codificación se refiere, el único paso restante en este ejemplo es modificar la clase Java que define la actividad, para que al iniciarse cargue la visual que hemos modificado. Para ello basta con modificar la única clase Java generada, `HelloAndroid.java`, y añadir una línea al método `onCreate()`, que es el método que se ejecuta cuando se crea la actividad (es decir, cuando se lanza la aplicación):

```
public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Asociamos nuestra visual a la actividad
        setContentView(R.layout.main);
    }
}
```

Con esta instrucción, le decimos a la actividad que su visual principal es la definida por el fichero `main.xml` del directorio `layout`.

5. **Ejecutar la aplicación**. Ahora ya podemos ejecutar la aplicación y observar el resultado. Para ello sólo tenemos que pulsar *Run as... Android application* en el menú contextual del proyecto. Si hemos definido bien el dispositivo Android virtual, Eclipse lo seleccionará automáticamente para ejecutar la aplicación, o bien nos pedirá que seleccionemos nosotros uno de los disponibles. El resultado debe ser algo similar a ésto:

Figura 2. Aplicación de ejemplo en ejecución.

Con esto finaliza la descripción y preparación del entorno utilizado, y podemos pasar a explicar la aplicación real objeto del proyecto: GUÍA UPCT.

La aplicación: GUÍA UPCT

A lo largo de este capítulo describiremos la aplicación GUÍA UPCT, tanto a nivel funcional general como entrando en el detalle técnico siempre que se considere necesario o de interés. Comenzaremos explicando la estructura general de la aplicación, sus actividades y flujo de funcionamiento, así como la distribución de las clases en paquetes, identificando la correspondencia entre cada una de las partes del patrón MVC (Modelo, Vista y Controlador) y dichas clases. A continuación iremos viendo cada una de las actividades que conforman la aplicación, sus pantallas y utilización, y las clases y recursos relacionados. Por último se hará un análisis de las fortalezas y carencias de la aplicación y mejoras que podrían aplicarse a nivel técnico.

Estructura general de la aplicación

La aplicación se divide en distintas actividades, las cuales se van llamando según las acciones llevadas a cabo por el usuario. Aunque no siempre es así en todas las aplicaciones Android, en nuestro caso se puede entender cada actividad como una pantalla diferenciada, una página (por utilizar un símil) desde la cual podemos llevar a cabo distintas acciones, consultar información o acceder a otra página distinta.

- **INICIO**. La primera pantalla es la de Inicio, es la pantalla de presentación de la aplicación y ofrece tres opciones: Mapa, Listado y Ayuda.

- **AYUDA**. La pantalla Ayuda muestra un texto describiendo la aplicación y sus principales funciones, y permite acceder a información adicional mediante el navegador web por defecto instalado en el dispositivo.

- **MAPA**. La pantalla Mapa muestra un mapa geográfico de Cartagena y, sobreimpresionados sobre el mismo, iconos representando cada uno de los edificios de la UPCT. Si se pulsa en alguno de estos iconos, aparecerá un diálogo que permite al usuario seleccionar una de las siguientes opciones: Mostrar Ruta, Ver Plano o volver al mapa. La primera opción muestra sobre el mismo mapa, el trazado de la ruta a pie más recomendada entre la ubicación actual del usuario y el edificio seleccionado; la segunda opción inicia la actividad Plano.

- **LISTADO**. La pantalla Listado muestra una lista de los edificios de la UPCT. Para cada edificio de la lista se muestra una pequeña fotografía, el nombre del edificio y el nombre de la entidad (Escuela y ógrano de gobierno) que alberga. Si se pulsa en cualquiera de los edificios de la lista se accede a la Ficha del mismo.

- **FICHA**. La pantalla Ficha muestra información ampliada sobre el edificio: el nombre y la descripción del edificio, entidad asociada y una fotografía más grande. Además permite seleccionar entre las opciones Mapa, Plano y Página web. Seleccionar la opción Mapa abrirá la actividad Mapa arriba descrita, pero

con el mapa centrado en el edificio seleccionado. Seleccionar la opción Plano iniciará la actividad Plano. Por último, seleccionar la opción Página web abrirá en el navegador por defecto la página web de la entidad de la UPCT asociada al edificio.

- **PLANO**. La pantalla Plano, finalmente, muestra un plano de la planta principal del edificio seleccionado. Este plano puede ser ampliado y desplazado por el usuario a conveniencia. Sobre dicho plano se muestran indicados los Puntos de Interés (POI) definidos para el edificio, como pueden ser la cafetería, el servicio de reprografía o la secretaría; estos puntos (marcados por un círculo rojo) pueden ser pulsados por el usuario, mostrándose en tal caso el nombre y descripción del POI y dando la opción de visitar la página web asociada.

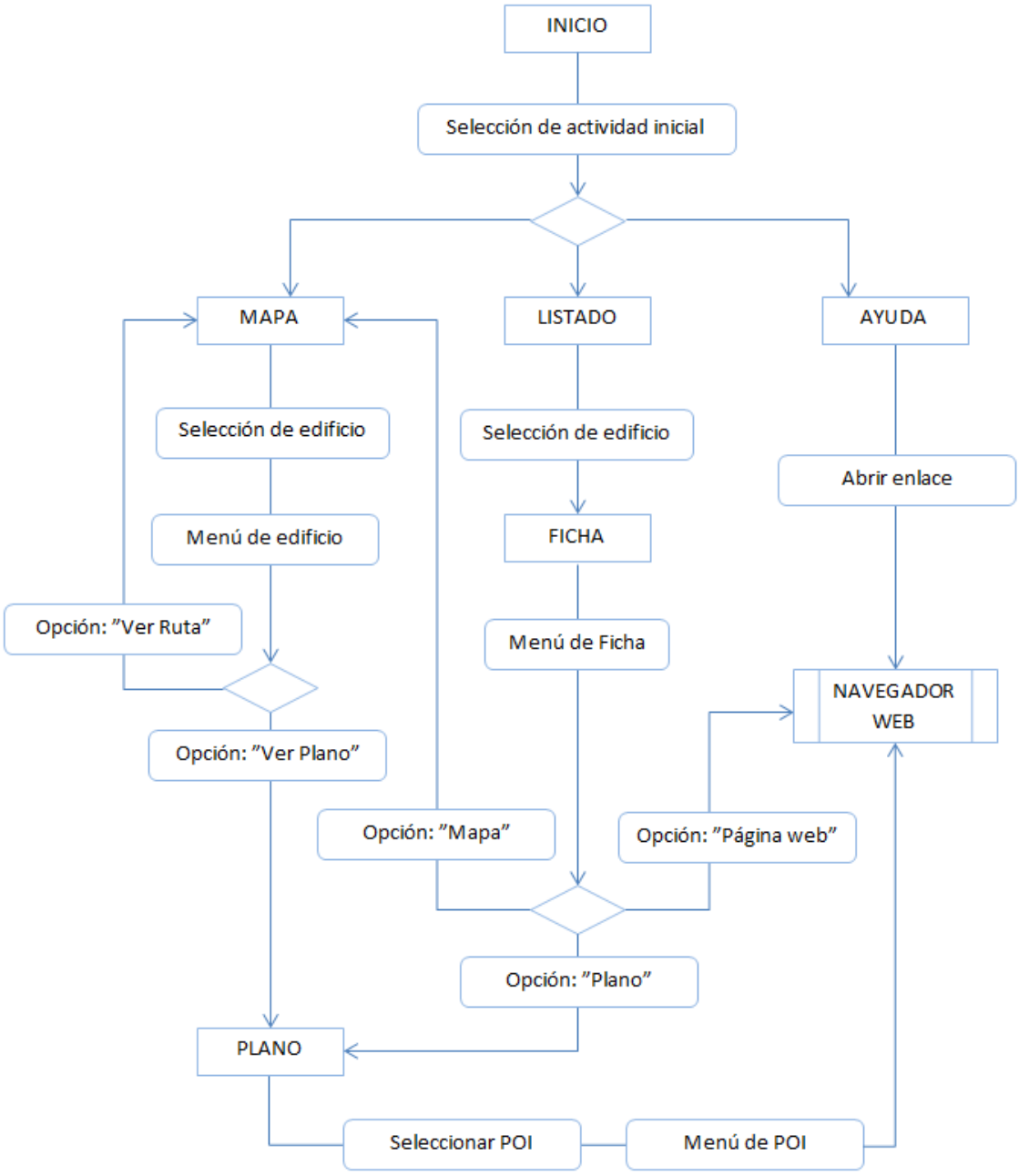

Figura 1. Grafo de navegación entre las distintas pantallas de la aplicación GUÍA UPCT.

Como en cualquier otra aplicación Android, pulsando repetidamente la tecla "Atrás" del dispositivo es posible regresar a la actividad anterior, así como abandonar definitivamente la aplicación en caso de que el usuario pulse la tecla Atrás desde la actividad inicial de la aplicación (la pantalla Inicio, en nuestro caso).

Organización clases y recursos

Como es normal y recomendable en cualquier proyecto Java, las clases de la aplicación GUÍA UPCT se han organizado en paquetes para una mejor identificación de sus roles. Como nombre "base" de la paquetización se ha elegido `org.proyecto.guiaupct` y a partir de dicho paquete raíz se han ido creando diferentes subpaquetes atendiendo a las características de las clases.

Para esta división en subpaquetes se podría haber elegido uno cualquiera de entre múltiples criterios, como por ejemplo el perfil de las clases dentro del patrón MVC: modelos, controladores y vistas; su relación con la API de Android: *activities*, *views*, *adapters*, *overlays*; o la funcionalidad a la que van asociadas: mapa, listado, ayuda. El ordenamiento elegido aúna cualidades de todos estos criterios, dividiendo las clases inicialmente por funcionalidad y posteriormente en base a su relación con la API de Android. Las clases relacionadas con el modelo de datos se han colocado en un paquete aparte, al mismo nivel que los de funcionalidades.

La paquetización queda como sigue:

- `org.proyecto.guiaupct`
  Paquete raíz de la aplicación. Alberga además la actividad inicial, `GUInicio.`
  - `org.proyecto.guiaupct.modelos`
    Clases del modelo de datos.
  - `org.proyecto.guiaupct.datos`
    Contiene la clase `DatosAplicacion`, que unifica el acceso al modelo de datos mediante la aplicación del patrón *Singleton.*
  - `org.proyecto.guiaupct.ayuda`
    Clases de la actividad Ayuda.
  - `org.proyecto.guiaupct.listado`
    Clases de las actividades Listado y Ficha.
    - `org.proyecto.guiaupct.listado.adapter`
      Clases de tipo `Adapter` utilizadas por las actividades Listado y Ficha.
  - `org.proyecto.guiaupct.mapa`
    Clases de las actividades Mapa y Plano.
    - `org.proyecto.guiaupct.mapa.kml`
      Clases utilizadas por la actividad Mapa para parsear ficheros KML.
    - `org.proyecto.guiaupct.mapa.overlays`
      Clases de tipo `Overlay` utilizadas por el `MapView` de la actividad Mapa.

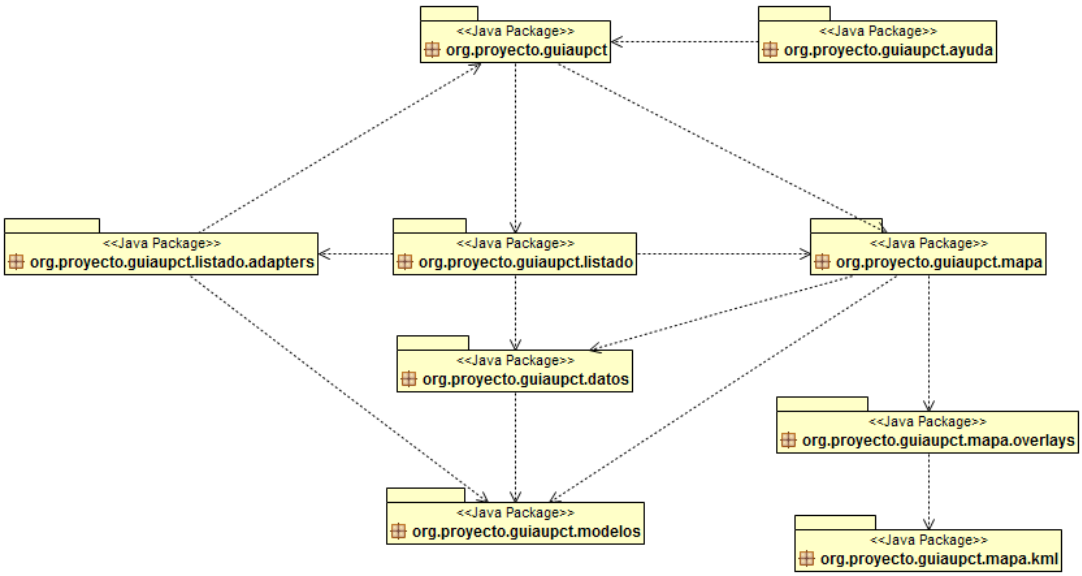

Figura 1. Diagrama de paquetes de la aplicación GUÍA UPCT, con las relaciones de dependencia entre los mismos.

En la siguiente página se muestra el diagrama de clases global de la aplicación, en el cual aparecen tanto las relaciones directas de tipo "asociación" como las indirectas de tipo "depende de". Las relaciones de tipo "generalización" (herencia) y "realización" (implementación de interfaces) se han ocultado para poder ofrecer un diagrama global más limpio, reservando ese nivel de detalle para diagramas más específicos situados en el apartado correspondiente a cada clase.

Cabe destacar que, aunque no se observe en el diagrama, todas las actividades principales (clases `GUMapa`, `GUPlano`, `GUFicha` y `GUListado`) obtienen los datos de los edificios a través de la clase `DatosAplicacion` y por tanto existe una relación de dependencia entre ésta y cada una de aquéllas.

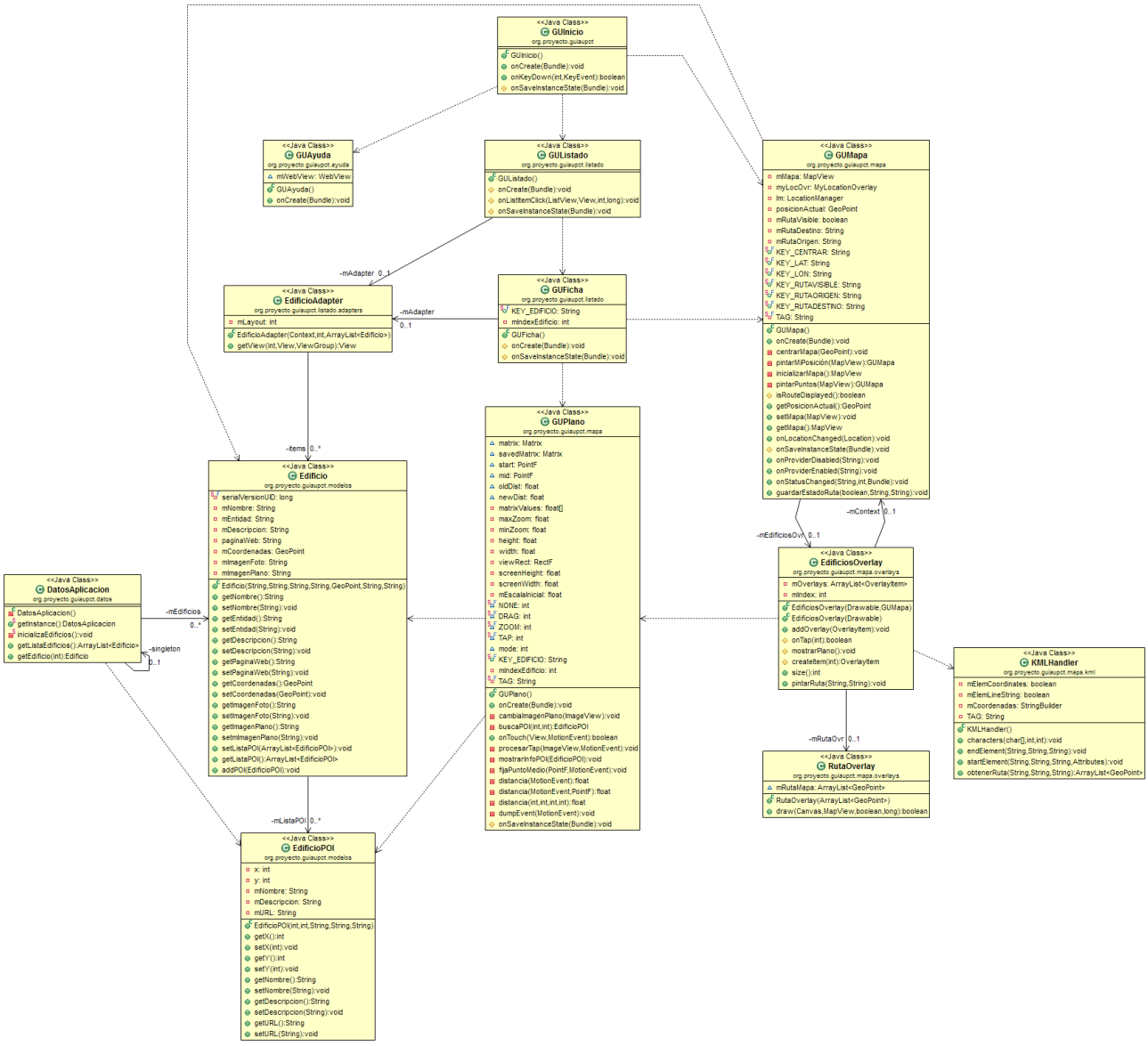

Figura 2. Diagrama de clases de la aplicación GUÍA UPCT.

Configuración y permisos

En el apartado dedicado al sistema operativo Android vimos que el acceso a las funcionalidades del sistema operativo se controla mediante permisos. Nuestra aplicación, para poder mostrar al usuario su posición y para poder calcular las rutas a los edificios, necesita acceso a la funcionalidad de localización, que nos permite a través de la API de Android utilizar elementos como los sensores GPS y WiFi para determinar con mayor o menor precisión la posición actual del dispositivo móvil. Además, tanto para descargar los mapas y la información de las rutas como para

permitir al usuario visualizar páginas web externas (sitios web de las escuelas, información de los POI), el usuario debe concedernos también permiso para que el dispositivo pueda acceder a internet en caso necesario.
Estos permisos son los únicos que hacen falta para que nuestra aplicación funcione sin restricciones; se muestra a continuación el código xml que los especifica:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
      android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission
      android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Diseñando el Modelo: Edificio y EdificioPOI

De los tres componentes del patrón MVC, el Modelo es el componente que representa la realidad, aquello que ya existe. Mientras que el Controlador y la Vista pueden ser diseñados y modificados a nuestro antojo para adaptarlos a nuestro objetivo, a cambios en el enfoque de la aplicación o a nuevas características, el Modelo por su naturaleza real y externa escapa de nuestro control, y nuestra única función es definir esta realidad lo mejor posible en términos de la tecnología utilizada, grabarla en piedra para poder utilizarla como base alrededor de la cual construir nuestro proyecto. Es, por tanto, una buena práctica diseñar las clases que conforman nuestro Modelo en las etapas más tempranas del proyecto; puesto que, una vez definidas y salvo error de análisis, es muy raro que sufran modificaciones, siendo más bien el Controlador quien deberá reformular su lógica y las Vistas las que deberán adaptar su aspecto, conforme a las peculiaridades del Modelo.

¿Cuál es nuestro Modelo, pues? ¿Cuál es la realidad que queremos presentar al usuario en una forma que le sea útil? ¿Qué aspecto del mundo real queremos modelar para que la lógica de nuestro programa pueda trabajar con su información y crear valor añadido? Analicemos nuestra aplicación: mapas, fichas, planos, listados. ¿Qué tienen en común?

Tomemos por ejemplo el mapa y el listado. El mapa, en la aplicación que nos ocupa, no deja de ser una presentación visual de los distintos edificios de la Universidad Politécnica de Cartagena, siguiendo un criterio de localización geográfica. El listado, a su vez, no deja de ser otro tipo de visualización de dichos edificios, en este caso ordenados según su identificador interno; pero la información que muestra corresponde a la misma realidad que presenta el mapa, es decir los edificios. Y edificios son lo que se muestra en las fichas de nuestra aplicación, en este caso de una forma que muestre al usuario la mayor información posible y en un formato cómodo para él. Podemos considerar, entonces, al Edificio como uno de los elementos de la realidad que queremos modelar.

Pero ¿qué hay del plano? En nuestra aplicación, un plano será una imagen estática, un mero lienzo sobre el que dibujar los puntos de interés de cada Edificio. El plano del

edificio no podemos considerarlo como un Modelo en sí mismo, ya que es un simple recurso; si acaso, podemos entenderlo como una propiedad más del Edificio que estamos representando. Sin embargo, lo que nosotros mostramos en el plano no son edificios, ni siquiera es un edificio en concreto: el plano muestra Puntos de Interés (POI), servicios que pueden resultar útiles al usuario dentro de un edificio concreto. Si comparamos el plano con el mapa, veremos que existe cierto paralelismo, en el sentido de que así como el mapa presenta la realidad de los edificios siguiendo un criterio de localización respecto de la ciudad, de igual manera el plano muestra al usuario otra parte de la realidad, los POI, siguiendo también un criterio de localización, en este caso respecto del edificio. También los POI tienen atributos propios similares a los de los edificios: nombre, descripción, página web... que son independientes del edificio en el que se encuadre el POI; de hecho, un Punto de Interés puede cambiar de situación, pueden aparecer nuevos POI o pueden desaparecer los ya existentes, sin que ello tenga relación directa con la realidad modelada por el Edificio. Vemos por tanto que los POI de un Edificio deben tener su propia entidad dentro de nuestro Modelo, aunque posteriormente sean tenidos en cuenta como parte de un Edificio.

Según todo lo anterior, nuestro Modelo lo forman las entidades Edificio y EdificioPOI. Este será el nombre de las clases Java que implementaremos y cuyo detalle veremos a continuación.

Clase Edificio

Esta clase modela un edificio del campus de la UPCT, con todos los atributos del mismo que nuestra aplicación necesita conocer para ofrecer las funcionalidades planteadas. Dichos atributos son los siguientes:

- `mNombre`: Nombre utilizado comúnmente para referirse al edificio (objeto `String`).
- `mDescripcion`: un pequeño texto que describe el edificio, su función actual, año de construcción, función anterior y año de restauración cuando aplique, etc. Modelado como una cadena de texto (objeto `String`).
- `mEntidad`: nombre de la entidad que hace uso del edificio, por ejemplo la Escuela de Turismo o el Rectorado. Modelado como un objeto `String.`
- `mCoordenadas`: coordenadas geográficas del edificio, necesarias para mostrar su posición en el mapa. Este atributo se modela como un objeto de tipo `GeoPoint`, más adecuado para este tipo de dato.
- `paginaWeb`: objeto de tipo `String` que contiene la URL de la página web asociada al edificio o, más exactamente, a la entidad que lo utiliza.
- `mImagenFoto`: cadena de texto (`String`) conteniendo el nombre del recurso que se utilizará como imagen descriptiva del edificio.
- `mImagenPlano`: cadena de texto (`String`) conteniendo el nombre del recurso

que se utilizará como plano del edificio.

- `mListaPOI`: Lista de Puntos de Interés (POI) que se encuentran dentro de este edificio. Modelada como un `ArrayList` de objetos `EdificioPOI.`

Además de los atributos anteriores, con sus correspondientes métodos `get()` y `set(),` la clase `Edificio` proporciona el método `addPOI(EdificioPOI)`, un método de conveniencia para llevar a cabo la asociación de objetos `EdificioPOI` con cada uno de los Edificios durante la inicialización de objetos de este tipo. Lo único que hace el método es añadir el objeto a la lista de puntos de interés del `Edificio`:

```
/**
 * Añade un nuevo POI a la lista de POI del edificio
 * @param poi el objeto <code>EdificioPOI</code> con la información del
POI
 */
public void addPOI(EdificioPOI poi) {
      getListaPOI().add(poi);
}
```

Clase EdificioPOI

La clase `EdificioPOI` representa un punto de interés que será dibujado por la aplicación sobre el plano de un edificio. Sus datos se definen mediante los siguientes atributos:

- `mDescripcion`: pequeña descripción del POI, la información más relevante. Tipo `String.`
- `mNombre`: objeto de tipo `String` con el nombre que identifica al punto de interés.
- `mURL`: cadena de texto (`String`) con la dirección URL de la página web que contiene la información ampliada del POI.
- `x, y`: de tipo entero (`int`), coordenadas X e Y del POI respecto de la imagen del plano en el que se dibuja.

Aparte de los métodos `get()` y `set()` de cada atributo, la clase `EdificioPOI` no define ningún método adicional.

Controladores y Vistas

Con lo que hemos visto en los capítulos correspondientes al sistema operativo Android y al patrón MVC, sabemos que la identificación de Controlador y Vista en una aplicación Android no es tan clara como en otros enfoques. Las visuales XML, aunque definen completamente la visual, siguen necesitando de las clases Java correspondientes para instanciar cada uno de los elementos declarados, con sus

propiedades y atributos correspondientes. Las clases Java de tipo `View`, por otra parte, en varias ocasiones no se limitan estrictamente a ofrecer una interfaz visual para el usuario, sino que añaden lógica auxiliar para facilitar el control de las mismas por parte del programador, a costa de traspasar la línea que separa la Vista del Controlador. En cuanto a las clases derivadas de `Activity`, según cómo se utilicen pueden asumir funciones propias de la capa Visual, añadiendo nuevos elementos a la interfaz de usuario según las necesidades de la aplicación y tomando el rol de *ViewModel* ya explicado.
Para definir un criterio único, en adelante identificaremos siempre como el Controlador del patrón MVC a las Actividades (clases derivadas de `Activity`), así como a cualquier otra clase auxiliar que no sea de tipo `View` y que ejecute lógica necesaria para llevar a cabo las funcionalidades de la aplicación. La Vista se corresponderá, por tanto, con las visuales XML y las clases derivadas de `View`, además de aquellas clases que sean utilizadas por éstas y cuya lógica se limite a actuar sobre la interfaz de usuario directa o indirectamente, sin llevar a cabo otro tipo de tareas como gestión de eventos o del ciclo de vida de la aplicación.
Así, las clases `GUMapa`, `GUPlano`, `GUInicio`, y todas las demás clases que heredan de `Activity` serían Controladores; la clase `KMLHandler`, que contiene la lógica para obtener y parsear las rutas del servicio de Google Maps, también formaría parte del Controlador de la actividad Mapa. La clase `EdificioAdapter`, en cambio, se consideraría parte de la Vista, pues su única función es servir de apoyo a las clases `View` instanciar unos componente visuales u otros según la visual XML que se esté utilizando. Lo mismo sucede con la clase `RutaOverlay`: sus métodos los invoca una clase de tipo `View` y toda la lógica que contiene afecta únicamente a la parte visual de la aplicación, sin llevar a cabo cálculos adicionales.
El caso de la clase `EdificiosOverlay` es especial; pues, aunque es utilizada por una instancia de `MapView` para dibujar los iconos de los edificios, también contiene cierta lógica de negocio necesaria para invocar las rutinas de obtención de la ruta. En general nos referiremos a esta clase indistintamente como parte del Controlador o de la Vista, según estemos hablando de su función específicamente visual o de su lógica de gestión.

Con estos criterios en mente, podemos pasar a analizar cada una de las pantallas y actividades de la aplicación GUÍA UPCT, entrando en el detalle de cada una de sus clases y visuales XML.

Primera Actividad: Mapa de edificios

Esta era, cuando se comenzó el desarrollo del proyecto, la primera y principal pantalla de la aplicación. Toda la idea de la GUÍA UPCT gira en torno a una aplicación Android que permita ubicar los distintos edificios de la UPCT en un mapa, y obtener de ellos información así como rutas para llegar a los mismos; esta actividad consigue precisamente eso.

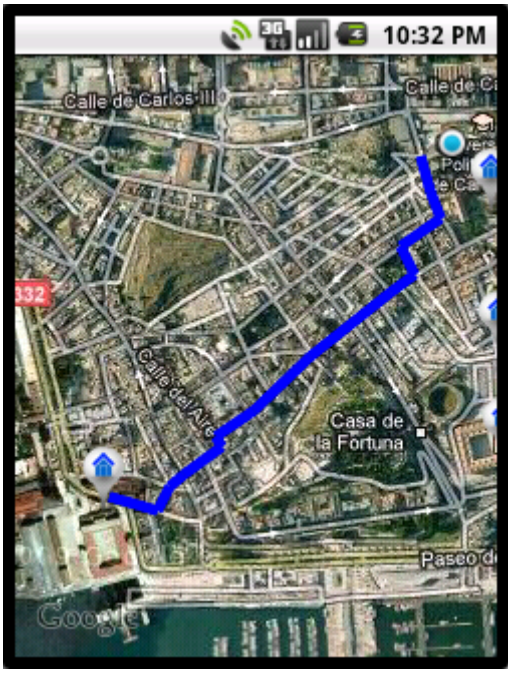

Figura 1. La actividad Mapa, mostrando la ruta a pie entre la posición actual del usuario y el edificio del antiguo C.I.M., sede de la Facultad de Ciencias de la Empresa.

Esta actividad se podría dividir en tres partes: por un lado, el mapa propiamente dicho, que es la base de la actividad y que se crea y gestiona usando las librerías de Google Maps para Android; por otro lado, la lógica relativa a los edificios, que se encarga de obtener la lista de edificios y de mostrar en pantalla marcadores para cada uno de ellos; y por último, cuando el usuario solicita que se muestre, la lógica que obtiene y muestra la ruta al edificio seleccionado.

Uso de las librerías de Google Maps

Cuando se visualiza en un navegador con una velocidad de conexión y procesamiento ideales, un mapa típico de Google Maps se ve como una única imagen que puede desplazarse en cualquier dirección, acercarse o alejarse a voluntad. En realidad el mapa está compuesto de múltiples imágenes cuadradas, pequeñas piezas que se descargan de manera independiente y que luego el navegador, mediante las funciones javascript de la API de Maps, compone para dar al usuario la sensación de una única gran imagen. De igual forma, cuando se hace zoom o se desplaza el mapa, lo que se

está haciendo no es un zoom directo sobre la imagen, ni tampoco hacer visibles partes de la imagen que estaban simplemente ocultas. Lo que hace la API es recalcular las coordenadas y el factor de acercamiento, y sustituir cada una de las piezas que ve el usuario por las imágenes adecuadas, las cuales pueden estar ya cacheadas por el navegador o puede que sea necesario volver a contactar con el servidor para descargarlas.

Figura 2. Detalle de un mapa de Google Maps en mitad de un aumento de zoom. Se puede ver con claridad el mosaico formado por la pieza inferior, que ya ha cargado la imagen a la resolución solicitada, y la pieza superior, que mantiene una versión ampliada de la imagen antigua.

Si se quisiera trasladar este sistema directamente a una actividad Android, sería necesario definir una visual compuesta de múltiples elementos de imagen alineados convenientemente, y un controlador que se encargue de manejar todas las rutinas descritas: descarga de las piezas, composición del mosaico de imágenes, control de la caché de piezas, lógica de desplazamiento y zoom, etc.

Afortunadamente, para facilitar a los desarrolladores el uso de mapas en sus aplicaciones, Google pone a nuestra disposición una librería externa de Google Maps que incluye el paquete `com.google.android.maps`; las clases de este paquete ofrecen de serie todas estas funcionalidades necesarias para mostrar un mapa de Google Maps, así como varios controles y opciones de presentación adicionales.

La clase principal del paquete es `com.google.android.maps.MapView`, de tipo visual. Una visual `MapView` muestra un mapa con datos obtenidos del servicio de Google Maps. Cuando la visual tiene el foco, captura los eventos del usuario y se encarga de desplazar y hacer zoom del mapa automáticamente, incluyendo las

solicitudes al servidor de nuevas piezas del mapa. También incluye todos los elementos de interfaz necesarios para que el usuario pueda controlar el mapa (controles de zoom, por ejemplo), y además ofrece al programador métodos que le permiten controlar el mapa programáticamente así como la posibilidad de dibujar *overlays*, "capas" a modo de transparencias para mostrar información sobre el mapa.

En resumen, la clase `MapView` proporciona un envoltorio para la API de Google Maps que permite a las aplicaciones Android manipular los datos de Google Maps a través de métodos Java, y trabajar con mapas de la misma manera que se haría con cualquier otro tipo de visual.

Hay que hacer notar, que la librería de Google Maps no forma parte de la biblioteca estándar de Android y por tanto puede no estar disponible en algunos dispositivos Android, aunque lo normal es que la mayoría de teléfonos móviles y tablets la incluyan junto con la propia aplicación "Maps" oficial de Google. De igual manera, la librería tampoco está incluida por defecto en el SDK. Puede obtenerse como parte de una extensión del SDK de Android, que incluye muchas de las API de Google y que puede descargarse gratuitamente.

Para poder mostrar las imágenes de los mapas en la visual `MapView`, es obligatorio además que el desarrollador se dé de alta en el servicio de Google Maps y obtenga una clave para utilizar la API. EL proceso de obtención de la misma es sencillo y puede consultarse en las referencias de esta memoria; no obstante, Google permite que los desarrolladores de Android utilicen en sus aplicaciones, durante la fase de pruebas al menos, una clave especial de depuración que da acceso a toda la funcionalidad de los mapas. Esta clave la genera el programador mediante los certificados que se incluyen en cada instalación del SDK de Android, y por tanto implica que el programa sólo funcionará si se compila con el mismo SDK con el que fue generada la clave, si bien la funcionalidad es completa. Su única limitación es que la aplicación no puede publicarse en el Android Market. La aplicación GUÍA UPCT hace uso de esta clave para facilitar el proceso de desarrollo.

Visual XML

Debido a que casi toda la funcionalidad del mapa está controlada por las librerías de Google Maps, la visual XML de la actividad Mapa es en realidad muy sencilla:

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.maps.MapView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/mapview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:clickable="true"
    android:apiKey="0fSvfw7Y8sudlw50zbdZOQaAd87i3L6Jpwz3DOA"
/>
```

Como vemos, lo único que hay que hacer para obtener un mapa funcional, es declarar un elemento de tipo `MapView` e indicar que podrá recibir eventos de tipo clic y, por último, pasarle la clave privada que hemos obtenido durante el proceso de configuración de la API de Google Maps.
Cabe destacar que, al ser el tipo `MapView` un elemento de una librería externa a Android (la API de Google Maps no está integrada en Android), es obligatorio referenciar el elemento XML por su nombre cualificado, es decir, incluyendo el nombre del paquete en el que se encuentra ubicada la clase.

Clase principal: GUMapa

Ya hemos visto que la visual del mapa, `MapView`, además de mostrar al usuario todo el contenido visual se encarga también de gestionar todas las tareas en segundo plano necesarias para descargar y organizar los mapas. La clase `GUMapa`, en este caso, puede liberarse de estas tareas y asumir un papel de controlador secundario o adicional, el rol de "presentador" que vimos en la sección dedicada a los patrones de diseño.
Así, la clase `GUMapa` se centra en controlar toda la lógica relacionada con funcionalidades específicas de la aplicación, como pueden ser el mostrar los edificios y las rutas, o comunicarse con el resto de actividades de la GUÍA UPCT. Para facilitar este desacoplamiento entre la visual y nuestro controlador, la clase `GUMapa` hereda de un tipo específico de actividad, `MapActivity`, que también viene incluido en la librería Maps y que está diseñado para utilizarse con visuales `MapView.`

Precisamente, una de las primeras operaciones que lleva a cabo la clase `GUMapa` es la inicialización de la visual del mapa y configuración de sus opciones:

```
private MapView inicializarMapa() {
      MapView mv = (MapView) findViewById(R.id.mapview);
      mv.setBuiltInZoomControls(true);
      mv.setSatellite(true);
      MapController mpc = mv.getController();

mpc.setCenter(DatosAplicacion.getInstance().getListaEdificios().get(0)
                  .getCoordenadas());
      mpc.setZoom(15);
      return mv;
}
```

Vemos que las opciones son varias y nos permiten controlar el tipo de mapa, cómo se presenta al usuario y las funcionalidades y controles disponibles. En nuestro caso indicamos las siguientes configuraciones:

- Mostrar controles de zoom: permite al usuario acercar o alejar el mapa mediante unos botones (+) y (-) en pantalla.
- Mostrar imágenes de satélite: por defecto, el mapa mostrado es sólo callejero;

activando esta opción conseguimos que el callejero se superponga sobre imágenes de satélite.

- Centrar mapa en el primer edificio: a través del mini-controlador que incorpora la visual, podemos instruir al mapa para que aparezca centrado en las coordenadas que nosotros deseemos. En este caso le indicamos que fije su centro en las coordenadas del primer edificio de la lista de edificios de la aplicación.
- Fijar zoom por defecto: los niveles de zoom del mapa van del 1 al 22, siendo 1 la vista más alejada y 22 el máximo zoom posible. Un zoom de nivel 15 permite que el mapa abarque una extensión suficiente como para que todos los edificios del campus aparezcan en pantalla.

Una vez creada y configurada la visual, retornamos el objeto que la representa para que la actividad pueda acceder a ella en caso necesario.

En este punto ya podríamos mostrar un mapa funcional, pero para que el usuario pueda sacarle partido nos hace falta tener acceso a las capacidades de ubicación del dispositivo. Anteriormente hemos visto cómo solicitar al usuario y al sistema operativo los permisos necesarios para tener acceso a las capacidades de geolocalización del dispositivo; sin embargo, por defecto una actividad Android no recibe información de geolocalización a menos que lo indique expresamente durante su ciclo de vida.
El siguiente paso, por tanto, es indicarle al sistema operativo que deseamos recibir actualizaciones de ubicación del dispositivo:

```
lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
lm.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 500L, 20.0f,
                          this);
lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 500L, 20.0f,
                          this);
```

Para ello utilizamos el método `requestLocationUpdates(...)` que nos ofrece el sistema operativo a través de la clase LocationManager. Este método acepta cuatro parámetros: el proveedor de la información, que puede ser la red GPS o las redes de telefonía y WiFi (en este caso usamos ambos); el tiempo mínimo entre actualizaciones; la distancia mínima que debe variar la posición para considerarlo una nueva ubicación; y el objeto de tipo LocationListener que va a recibir las actualizaciones de posicionamiento.
Vemos que el último parámetro que le estamos pasando a dicho método es `this`; de esta manera, nuestra actividad se convierte en receptora directa de las actualizaciones. Para ello, la clase `GUMapa` implementa la interfaz `LocationListener`, la cual contiene los métodos que el sistema operativo ejecutará cada vez que quiera informar de cambios en la ubicación o en el estado de los sistemas de posicionamiento.
El método más importante de esta interfaz es `onLocationChanged(...)`, llamado cada vez que varía la ubicación del dispositivo:

```
@Override
public void onLocationChanged(Location location) {
      if (location != null) {
            double lat = location.getLatitude() * 1E6;
            double lon = location.getLongitude() * 1E6;
            posicionActual = new GeoPoint((int) lat, (int) lon);
            Log.d(TAG, "Nueva posición");
      }
}
```

En caso de recibir nuevas coordenadas, las guardamos para su posterior uso.

Puesto que ya disponemos de un mapa funcional y de la capacidad de obtener nuestra posición actual para calcular la ruta a los edificio, el siguiente paso lógico es mostrar al usuario la posición de esos edificios para que pueda seleccionar a cuál dirigirse, si lo desea.
Para acometer esta tarea, la clase `GUMapa` hace uso de la clase `EdificiosOverlay`, que veremos en detalle más adelante. De momento, baste decir que es un objeto de tipo `Overlay` que nos permitirá añadirle al mapa una capa en la cual poder marcar mediante iconos la posición de nuestros edificios.
La lógica que sigue es la siguiente:

```
private GUMapa pintarPuntos(MapView mv) {

      // Cada edficio se marcará con este icono...
      Drawable drawable =
this.getResources().getDrawable(R.drawable.marcador);
      // ... en esta nueva capa...
      mEdificiosOvr = new EdificiosOverlay(drawable, this);
      // ... y representado por este objeto...
      OverlayItem overlayItem;
      // ... que se pintará en estas coordenadas.
      GeoPoint punto = null;

      // Para cada uno de los edificios
      for (Edificio e :
DatosAplicacion.getInstance().getListaEdificios()) {
            // Recuperamos sus coordenadas,
            punto = e.getCoordenadas();
            // creamos un nuevo item con sus datos,
            overlayItem = new
OverlayItem(punto,e.getNombre(),e.getEntidad());
            // y lo añadimos a la capa de edificios
            mEdificiosOvr.addOverlay(overlayItem);
      }

      // Finalmente recuperamos la lista de capas del mapa
      List<Overlay> mapOverlays = mv.getOverlays();
      // y añadimos la nuestra
      mapOverlays.add(mEdificiosOvr);

      return this;
}
```

El primer bloque crea e inicializa los objetos necesarios. Un objeto `Drawable` representa cualquier imagen o recurso que se pueda dibujar en una visual; en nuestro caso, utilizamos un `Drawable` para cargar el icono que marcará la posición de cada uno de los edificios. El objeto `EdificiosOverlay`, como ya hemos dicho, representa la capa que se va a añadir a la visual para dibujar sobre ella. Finalmente, creamos un objeto `OverlayItem` que va a representar cada uno de los elementos que pintaremos en la nueva capa.

El segundo bloque va creando cada uno de estos elementos. Obtiene y recorre la lista de edificios de la aplicación y, para cada uno de ellos, crea un nuevo objeto `OverlayItem` conteniendo las coordenadas del edificio, su nombre, y el nombre de la escuela u órgano de la Universidad que utiliza dicho edificio. Cada uno de estos objetos es incorporado a la nueva capa mediante el método `addOverlay(OverlayItem)`.

Por último sólo tenemos que añadir nuestro *overlay* recién creado a la lista de *overlays* de la visual de Google Maps. Será esta la que se encargue de recuperar los item de cada capa, entre ellas nuestro `EdificioOverlay`, e ir pintando el icono definido para la capa en las coordenadas de cada item.

Cuando se accede desde la pantalla de Inicio, la última rutina que lleva a cabo la clase `GUMapa` es la de marcar la posición del propio usuario. Si quisiéramos implementar nosotros mismos la lógica necesaria para llevar a cabo esta tarea, el procedimiento sería similar al que hemos seguido para dibujar la ubicación de los edificios: crear una clase que herede de `Overlay`; adaptar sus constructores y métodos para que sea capaz de posicionar un icono en las coordenadas del usuario; y añadir una instancia de la misma a la lista de capas de la visual.

Por suerte, la librería de Google Maps para Android tiene en cuenta que esta es una de las funcionalidades más demandadas en un mapa, y para ello incorpora en la misma librería un *overlay* creado específicamente para eso: `MyLocationOverlay.` Esta clase hereda de `Overlay`, como es normal, e incluye de serie rutinas que, automáticamente, calculan la posición del dispositivo y dibujan no sólo dicha posición sino también la precisión del cálculo y, opcionalmente, una pequeña brújula para facilitar la orientación.

Con `MyLocationOverlay` a nuestra disposición, lo único que tenemos que hacer es crear una instancia de la clase, configurarle algunos parámetros y añadirla a la visual como otra capa cualquiera:

```
private GUMapa pintarMiPosición(MapView mv) {
      myLocOvr = new MyLocationOverlay(this, mv);
      myLocOvr.enableMyLocation();
      mv.getOverlays().add(myLocOvr);
      myLocOvr.runOnFirstFix(new Runnable() {
            public void run() {
                  posicionActual = myLocOvr.getMyLocation();
            }
      });
      return this;
}
```

La parte más destacable de esta inicialización es la utilización del método `runOnFirstFix(Runnable)`. El método acepta como parámetro un objeto de tipo `Runnable`, que a efectos prácticos viene a ser una rutina que se pueda ejecutar en un hilo aparte, y su función es permitir al usuario determinar la lógica que debe seguir la instancia de `MyLocationOverlay` la primera vez que consiga determinar la ubicación del dispositivo.

En nuestro caso hemos creado un objeto `Runnable` cuya rutina principal sirve para almacenar la información de la posición calculada. Al pasarle este objeto al método `runOnFirstFix(...)`, conseguimos que el *overlay* nos envíe la información de las primeras coordenadas obtenidas, las guardamos y así podemos usarlas más tarde para calcular las rutas.

Clase auxiliar: EdificiosOverlay

Hemos dicho que la clase `EdificiosOverlay` es utilizada por la clase `GUMapa` como una capa transparente, en la que dibujar sobre el mapa la posición de los edificios de la UPCT. Esta descripción, no obstante, no deja de ser una simplificación del cometido de esta clase; `EdificiosOverlay` no sólo muestra los marcadores de los edificios, sino que se encarga de gestionar toda la lógica relacionada con ellos:

- Recibe los eventos de pulsación sobre los marcadores.
- Interacciona con el usuario, mostrándole la información de cada marcador y las opciones disponibles.
- Ejecuta todos los pasos necesarios para, mediante clases auxiliares, obtener y dibujar la ruta al edificio seleccionado en caso de que el usuario lo solicite.
- Lanza la actividad Plano si ésta es la opción pedida por el usuario.

Por tanto la clase `EdificiosOverlay` es en realidad un complemento de la clase `GUMapa`, tan importante como ésta, y aunque mantiene el sufijo `Overlay` por convención en realidad forma parte de lo que sería el controlador en MVC.

Los métodos más importantes de la clase `EdificiosOverlay` son tres: el que muestra las opciones al usuario cuando éste pulsa uno de los marcadores; el que se encarga de mostrar la ruta; y el que se encarga de lanzar la actividad Plano.

El primer método es de tipo evento (ejecutado automáticamente cuando la visual detecta una acción del usuario). Lo primero que hace la lógica del evento es recuperar el objeto `OverlayItem` asociado a la acción del usuario. Anteriormente vimos que la clase `GUMapa` recorría la lista de edificios para ir generando objetos `OverlayItem` que posteriormente añadía a la colección de objetos de `EdificiosOverlay.` Al pulsar el usuario uno de los marcadores, el evento `onTap(int)` recibe un entero que indica precisamente cuál de todos esos `OverlayItem` es el asociado.

Una vez tenemos esta información, la clase `EdificiosOverlay` crea un diálogo estándar de tres botones, que mostrará al usuario el nombre del edificio y escuela seleccionados y, a su vez, le permitirá elegir entre ver la ruta al edificio, ver el plano del mismo, o cancelar la acción. A cada uno de los botones del diálogo se le asocia un manejador de eventos creado *ad hoc* mediante una clase anónima. Así, si el usuario pulsa el primer botón, se llamará al método `pintarRuta(...)` pasándole las coordenadas de origen y destino de la ruta solicitada; si pulsa el segundo, se llamará a `mostrarPlano()`; y si se pulsa el último, simplemente se cierra el diálogo.

En la siguiente página se muestra la lógica del método para una mejor comprensión.

```
@Override
protected boolean onTap(int index) {
	mIndex = index;
	final OverlayItem item = mOverlays.get(index);
	AlertDialog.Builder dialog = new AlertDialog.Builder(mContext);
	dialog.setTitle(item.getTitle());
	dialog.setMessage(item.getSnippet());
	dialog.setPositiveButton("Ver Ruta", new OnClickListener() {
		public void onClick(DialogInterface dialog, int which) {
			dialog.dismiss();
			String destino = coords(item.getPoint());
			String origen = "";
			if (mContext.getPosicionActual() != null) {
				origen = coords(mContext.getPosicionActual());
			}
			pintarRuta(destino, origen);
		}
	});
	dialog.setNeutralButton("Ver Plano", new OnClickListener() {
		public void onClick(DialogInterface dialog, int which) {
			mostrarPlano();
			dialog.dismiss();
		}
	});
	dialog.setNegativeButton("Atrás", new OnClickListener() {
		public void onClick(DialogInterface dialog, int which) {
			dialog.dismiss();
		}
	});
	dialog.show();
	return true;
}
```

El tercer método, que se encarga de iniciar la actividad Plano, es bastante sencillo, pues su lógica no va más allá de crear el `Intent` asociado a la actividad, pasarle la información del edificio seleccionado (que en este caso, al ser el índice de la lista de edificios, coincide con el índice del objeto `OverlayItem` seleccionado por el usuario), y solicitar el inicio de la actividad:

```
protected void mostrarPlano() {
	Intent i = new Intent(mContext, GUPlano.class);
	i.putExtra(GUPlano.KEY_EDIFICIO, mIndex);
	mContext.startActivity(i);
}
```

Solo resta por explicar el método `pintarRuta(...)`, encargado de controlar la

obtención de los datos de la ruta y dibujo de la misma sobre el mapa.

```
public void pintarRuta(String destino, String origen) {
      if (!("").equals(origen)) {
            if (mRutaOvr != null) {
                  mContext.getMapa().getOverlays().remove(mRutaOvr);
            }
            KMLHandler kh = new KMLHandler();
            ArrayList<GeoPoint> listgp = kh.obtenerRuta(origen, destino,
"w");
            mRutaOvr = new RutaOverlay(listgp);
            mContext.getMapa().getOverlays().add(mRutaOvr);

      mContext.getMapa().getController().animateTo(coords(origen));
            mContext.guardarEstadoRuta(true, destino, origen);
      }
      else {
            Toast.makeText(mContext,
                        "No puede mostrarse la ruta porque no se ha
podido"
                        +"determinar la posición actual.",
                        Toast.LENGTH_LONG)
                  .show();
      }
}
```

En primer lugar se verifica que las coordenadas de origen son válidas. En tiempo de ejecución puede suceder que no se consiga determinar la ubicación del usuario, en cuyo caso no se podría calcular ninguna ruta al no disponer deun punto de origen del que partir. Si se da esta situación, la aplicación muestra un mensaje informativo al usuario. Si las coordenadas son válidas, el método comprueba si ya se ha generado alguna ruta y en tal caso elimina el *overlay* asociado de la lista de *overlays* del mapa, borrando efectivamente cualquier ruta de la pantalla.

A continuación `EdificiosOverlay` se sirve de la clase auxiliar `KMLHandler` para obtener la lista de los puntos geográficos que componen la ruta deseada. La clase `KMLHandler` hace uso de los servicios de Google Maps para obtener dicha ruta y se verá con mayor profundidad más adelante.
Con dicha lista de puntos, `EdificiosOverlay` se sirve de otra clase de tipo `Overlay` para crear una nueva capa sobre la que dibujar la ruta: la clase `RutaOverlay.` Una vez creada, esta nueva capa se añade a la lista de capas de la visual `MapView` para que se dibuje junto con el resto de capas (los edificios y la posición del usuario).

Para terminar, el método reposiciona el mapa para centrarlo en el origen de la ruta, es decir, la posición de partida en la que se encuentra el usuario; y guarda cierta información de estado, que indica que hay una ruta visible, y que permitirá

redibujarla en caso de un cambio de orientación de la pantalla, por ejemplo.

Clase auxiliar: RutaOverlay

`RutaOverlay` es una clase de tipo `Overlay` igual que `EdificiosOverlay` pero, a diferencia de ésta, `RutaOverlay` puede considerarse un `Overlay` "puro" en dos sentidos:

- Al no requerir ningún tipo de interacción con el usuario, no incorpora lógica de control de la aplicación, limitándose a llevar a cabo las rutinas gráficas necesarias para dibujar en pantalla la ruta solicitada.

- Mientras que `EdificiosOverlay` heredaba de `ItemizedOverlay` para poder utilizar objetos con información añadida, `RutaOverlay` extiende directamente de `Overlay` porque lo único que necesita para pintar la ruta es un lienzo sobre el que dibujar los segmentos que la forman.

La estrategia que sigue `RutaOverlay` para pintar la ruta (ruta que ha recibido como parámetro del constructor) consiste en sobrescribir directamente el método `draw(...)`, que es el método que ejecuta `MapView` sobre cada una de sus capas para que éstas se dibujen a sí mismas. El código de este método es como sigue:

```
@Override
public boolean draw(Canvas canvas, MapView mapView, boolean shadow,
            long when) {

      /* Referencias a las rutas y puntos */
      ArrayList<GeoPoint> ruta = mRutaMapa;
      Point p1 = new Point();
      Point p2 = new Point();

      /* Configuración de la línea que pintaremos para cada segmento de
ruta */
      Paint linea = new Paint();
      linea.setStrokeWidth(5);
      linea.setColor(Color.BLUE);
      linea.setAntiAlias(true);

      /* Transformamos los GeoPoint del mapa a Point del lienzo (canvas)
       * y vamos pintando los segmentos para cada par de puntos
       */
      Projection proyección = mapView.getProjection();
      Iterator<GeoPoint> it = ruta.iterator();
      if (it.hasNext()) {
            proyección.toPixels(it.next(), p1);
            while (it.hasNext()) {
                  proyección.toPixels(it.next(), p2);
                  canvas.drawLine(p1.x, p1.y, p2.x, p2.y, linea);
                  p1 = new Point(p2);
            }
      }
      return super.draw(canvas, mapView, shadow, when);
}
```

Como vemos, el método `draw(...)` recorre la lista de puntos de la ruta mediante un iterador y, a través de la clase `Projection` que nos provee la visual `MapView`, obtiene una proyección de los puntos sobre el mapa, una traducción entre las coordenadas geográficas de cada punto y el píxel que las representa en el lienzo del mapa. Una vez hecho esto, para pintar la ruta se va dibujando en el lienzo del *overlay* un segmento entre cada uno de los puntos y el siguiente, mediante el método `drawLine(...)`, hasta llegar al último.

Obtención de la ruta: clase KMLHandler

Para obtener la lista de puntos que componen la ruta solicitada por el usuario, la clase `EdificiosOverlay` llama al método `obtenerRuta(...)` de la clase auxiliar `KMLHandler`. Para comprender cómo obtiene esta clase dicha información, vamos a ayudarnos del comentario Javadoc asociado a la clase:

```
/**
 * Clase que accede a la API de Google Maps vía web para recuperar
 * la ruta entre dos puntos, en formato KML (derivado de XML).
 * Implementa algunos métodos del <i>parser</i> XML nativo de Android,
 * SAXParser, para poder extraer la información del tag KML adecuado.
 */
public class KMLHandler extends DefaultHandler {
```

Igual que el comentario, la funcionalidad de la clase puede dividirse en dos partes. Por una parte, la clase `KMLHandler` se encarga de contactar con el servicio de Google Maps para obtener la ruta óptima entre los puntos origen y destino. Pero ¿no se supone que somos nosotros los que calculamos las rutas?

Como se explica en las motivaciones del proyecto, uno de los objetivos del mismo es demostrar que se pueden desarrollar aplicaciones útiles combinando la funcionalidad de servicios ya existentes para crear valor añadido. Es cierto que, con mucho mayor tiempo y esfuerzo, podríamos haber desarrollado nosotros mismos un software que realmente "calculara" la ruta: interpretando los trazos de las calles que aparecen en el mapa de Google Maps, generando grafos que representen dichas calles y aplicando los algoritmos adecuados podemos hallar la ruta más corta entre dos vértices.

Sin embargo, ¿no es una de las máximas de la programación el no reinventar la rueda cuando no es necesario? Si disponemos ya de un servicio que calcula la ruta entre dos puntos, ¿por qué no usarlo? Es por este motivo que nuestra aplicación no calcula las rutas directamente, sino que a través de la clase `KMLHandler` las obtiene del servicio web de Google Maps, una opción mucho más eficiente y fiable.

Para acceder desde nuestra clase Java, via web, al servicio de Google Maps y obtener del mismo la ruta en un formato que nos permita analizarla posteriormente, el código que consigue esto es el siguiente (nota: para mayor claridad se ha eliminado código no relacionado directamente con el acceso al servicio web):

```
String urlServicioKML =
      "http://maps.google.es/maps?f=d&hl=en"
      + "&saddr=" + origen + "&daddr=" + destino + "&dirflg=" + modo
      + "&ie=UTF8&0&om=0&output=kml";

Log.d(TAG, "URL " + urlServicioKML);
URL url = null;
// (...)
XMLReader xr = null;

try {
      url = new URL(urlServicioKML.toString());
      // (...)
      xr.parse(new InputSource(url.openStream()));
} catch (Exception e) {
      Log.e(this.TAG, e.toString());
}
```

El primer paso es formar la URL de acceso al servicio web. Aparte del origen y el destino, los parámetros que necesitamos pasarle al servicio para obtener la ruta pueden consultarse online en la documentación de referencia de la API de Google Maps (ver bibliografía). Vamos a destacar únicamente uno de ellos: `&output=kml`.

Mediante este parámetro, solicitamos al servicio que nos devuelva la información de la ruta en formato KML. KML (del acrónimo en inglés *Keyhole Markup Language*) es un lenguaje de marcado basado en XML que se utiliza para representar datos geográficos en tres dimensiones, y es utilizado por multitud de dispositivos (comerciales o no) de navegación GPS para exportar información de ruta o interpretar datos de navegación externos.

Si hicéramos una solicitud por defecto, Google Maps nos devolvería una página web normal y corriente que contendría la información necesaria para que un navegador web pueda mostrar un mapa con la ruta. A nuestra aplicación, en cambio, esa información no le sirve: lo que nosotros necesitamos es recibir la información de los puntos que componen la ruta, en un formato de tipo texto que podamos analizar. KML, al ser un lenguaje basado en XML, es fácil de interpretar; y al ser un formato extendido, existe multitud de documentación sobre cómo hacerlo. Estos dos motivos lo hacen un formato ideal para utilizarlo en cualquier desarrollo que necesite analizar rutas de navegación, como es el nuestro.

En la siguiente página puede verse un ejemplo del contenido de este tipo de ficheros.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.0">
<Document>
<name>Av de Trovero Marín/N-332 to Calle Real/N-332</name>
(...)
<Placemark>
<name>Head north on Av de Trovero Marín/N-332 toward Calle Ciudad de
Oran</name>
<description>
<![CDATA[go 44 m]]>
</description>
<address>Av de Trovero Marín/N-332</address>
(...)
<Point>
<coordinates>-0.977440,37.603290,0</coordinates>
</Point>
(...)
</Placemark>
(...)
<LineString>
<coordinates>-0.977440,37.603290,0.000000 -0.977530,37.603680,0.000000
(...) -0.987340,37.599010,0.000000 </coordinates>
</LineString>
(...)
</Document>
</kml>
```

A estas alturas ya tenemos a nuestra disposición toda la información de la ruta en un formato de texto estructurado y fácil de analizar, pero nos falta precisamente eso: analizarlo, pues la correspondencia no es directa. Aquí es donde entra en liza la segunda parte de la funcionalidad de la clase `KMLHandler`.

Un *parser* es un programa o proceso que es capaz de analizar un texto estructurado e identificar los elementos que lo componen; por ejemplo, los navegadores web contienen un *parser* que analiza el código HTML de las páginas para identificar los elementos que componen la página y sus interrelaciones, de igual forma que los compiladores contienen un *parser* que analiza el código fuente de un programa para extraer la secuencia de instrucciones que contiene en el orden correcto. En nuestro caso, Android contiene la librería SAXParser, un sencillo *parser* que puede interpretar una variedad de formatos, entre ellos XML, y que puede ser utilizado en cualquier desarrollo que necesite esta funcionalidad.

Nosotros necesitamos un *parser* para analizar el código KML recibido del servicio de Google Maps y extraer la lista de coordenadas que conforman nuestra ruta. KML se parece lo bastante a XML como para que SAXParser pueda interpretarlo con su *parser* XML por defecto; sin embargo, SAXParser no puede saber qué es lo que queremos

hacer nosotros con la información de cada elemento. Es por eso que para analizar correctamente el fichero KML, SAXParser necesita un *handler* o "manejador", una clase auxiliar que se encargará de recibir cada uno de los elementos detectados por el *parser* para examinar su contenido y actuar en consecuencia.

La clase de la que hereda `KMLHandler`, `DefaultHandler`, es una clase que proporciona métodos de interacción con SAXParser. Al extender esta clase, `KMLHandler` hereda los métodos que el parser XML de SAXParser utilizará para informar del comienzo, el contenido y el fin de cada elemento KML detectado. Nosotros sólo tenemos que implementar nuestra propia versión de manera que sea capaz de extraer la información del elemento `coordinates.`

El primer método que vamos a ver es el método `characters(...)`, llamado por SAXParser cada vez que detecta contenido útil dentro de un elemento (es decir, que no contiene más elementos):

```
@Override
public void characters(char[] ch, int start, int length) throws
SAXException {
      if (mElemLineString && mElemCoordinates) {
            mCoordenadas.append(ch, start, length);
            Log.d(this.TAG, mCoordenadas.toString());
      }
}
```

En nuestro caso, sólo nos interesa el contenido del último elemento `coordinates.` Si consultamos el ejemplo de fichero KML, el último elemento `coordinates` se diferencia del resto en que está englobado por un elemento `LineString` en vez de por un elemento `Point`; por lo que en este método sólo tenemos que añadir una comprobación basada en dos indicadores `mElemLineString` y `mElemCoordinates`, de manera que en `mCoordenadas` se guarde únicamente el contenido del elemento `coordinates` que buscamos.

¿Y cuándo se activan los indicadores? De eso se encargan los siguientes métodos, `startElement(...)` y `endElement(...)`:

```
@Override
public void startElement(String uri, String localName, String qName,
            Attributes attributes) throws SAXException {
      Log.d(this.TAG, "start " + localName);
      if ("LineString".equals(localName)) {
            mElemLineString = true;
      } else if ("coordinates".equals(localName)) {
            mElemCoordinates = true;
      }
}
```

El método `startElement(...)` es llamado por SAXParser cada vez que comienza un elemento. Un fichero KML puede tener cientos o miles de elementos distintos, pero a nosotros sólo nos interesan dos: el elemento `coordinates` y el elemento `LineString`. Cuando el *parser* nos indique que hemos entrado en un elemento con cualquiera de esos dos nombres, deberemos activar el indicador correspondiente.

```
@Override
public void endElement(String uri, String localName, String qName)
            throws SAXException {
      Log.d(this.TAG, "end " + localName);
      if ("LineString".equals(localName)) {
            mElemLineString = false;
      } else if ("coordinates".equals(localName)) {
            mElemCoordinates = false;
      }
}
```

De igual forma, el método `endElement(...)` se llama desde SAXParser cada vez que se detecta el fin de un elemento. Nosotros lo utilizamos para controlar que los indicadores `mElemLineString` y `mElemCoordinates` no permanezcan activos siempre, evitando que el método `characters(...)` lea contenido incorrecto; así, cuando el parser detecte que finaliza uno cualquiera de los elementos `LineString` o `coordinates`, pondremos a `false` su indicador correspondiente.

Con estos tres métodos implementados, tan sólo tenemos que indicarle a SAXParser que los encargados de manejar los eventos que genere durante el "parseo" del fichero KML vamos a ser nosotros mismos. Mostramos, ahora sí, el código completo para mayor claridad:

```
String urlServicioKML = "http://maps.google.es/maps?f=d&hl=en"
      + "&saddr=" + origen + "&daddr=" + destino + "&dirflg=" + modo
      + "&ie=UTF8&0&om=0&output=kml";

Log.d(TAG, "URL " + urlServicioKML);
URL url = null;
SAXParserFactory spf = null;
SAXParser sp = null;
XMLReader xr = null;

try {
      url = new URL(urlServicioKML.toString());
      spf = SAXParserFactory.newInstance();
      sp = spf.newSAXParser();
      xr = sp.getXMLReader();
      xr.setContentHandler(this);
      xr.parse(new InputSource(url.openStream()));
} catch (Exception e) {
      Log.e(this.TAG, e.toString());
}
```

Vemos cómo al método `setContentHandler(ContentHandler)` del parser XML le pasamos `this` como parámetro, indicando que los eventos los recibirá la misma clase `KMLHandler`.

La secuencia que sigue el proceso de "parseo" del fichero KML sería la siguiente:

1. Comienza el tratamiento del fichero. SAXParser detecta la apertura de un elemento y notifica a `KMLHandler` mediante llamada al método `startElement(...)`. `KMLHandler` comprueba que el elemento no se corresponde con ninguno de los dos que nos interesan y por tanto no lleva a cabo ninguna acción.
2. SAXParser lee el contenido de un elemento y lo transmite a `KMLHandler` mediante llamadas al método `characters(...)`. `KMLHandler` comprueba que no hay ningún indicador habilitado y desecha la información recibida.
3. SAXParser detecta el cierre de un elemento e informa a `KMLHandler` mediante llamada a `endElement(...)`. KMLHandler comprueba que el elemento no corresponde a los esperados y no lleva a cabo ninguna acción.
4. Los pasos anteriores se repiten para cualesquiera elementos que no coincidan con los esperados, y en cualquier orden: pueden sucederse varias aperturas sin ningún cierre y viceversa, puede que ningún elemento tenga contenido útil, etc.
5. SAXParser detecta la apertura de un elemento `coordinates` y notifica a `KMLHandler` mediante llamada al método `startElement(...)`. `KMLHandler` activa el indicador `mElemCoordinates` de manera acorde. El indicador `mElemLineString` permanece desactivado.
6. SAXParser lee el contenido del elemento `coordinates` y lo transmite a

`KMLHandler` mediante llamadas al método `characters(...)`. `KMLHandler` comprueba que aunque el indicador `mElemCoordinates` está activado, el indicador `mElemLineString` sigue desactivado; por tanto, este no es el elemento `coordinates` que buscamos y la información recibida se desecha.

7. SAXParser detecta el cierre del elemento `coordinates` e informa a `KMLHandler` mediante llamada a `endElement(...)`. `KMLHandler` desactiva el indicador `mElemCoordinates.`
8. Los pasos anteriores se repiten un número indeterminado de veces.
9. SAXParser detecta la apertura de un elemento `LineString` y notifica a `KMLHandler` mediante llamada al método `startElement(...)`. `KMLHandler` activa el indicador `mElemLineString` de manera acorde. El indicador `mElemCoordinates` permanece desactivado.
10. SAXParser detecta la apertura de un elemento `coordinates` y notifica a `KMLHandler` mediante llamada al método `startElement(...)`. `KMLHandler` activa el indicador `mElemCoordinates` de manera acorde. El indicador `mElemLineString` permanece activado.
11. SAXParser lee el contenido del elemento `coordinates` y lo transmite a `KMLHandler` mediante sucesivas llamadas al método `characters(...)`. `KMLHandler` comprueba que ambos indicadores están activados; por tanto, éste sí es el elemento `coordinates` que buscamos. La información recibida se almacena en un objeto `StringBuilder` que permite concatenar arrays de caracteres.
12. SAXParser detecta el cierre del elemento `coordinates` e informa a `KMLHandler` mediante llamada a `endElement(...)`. `KMLHandler` desactiva el indicador `mElemCoordinates.`
13. SAXParser detecta el cierre del elemento `LineString` e informa a `KMLHandler` mediante llamada a `endElement(...)`. `KMLHandler` desactiva el indicador `mElemLineString.`

Llegados a este punto, tenemos en nuestro objeto de tipo `StringBuilder` todo el contenido del elemento `coordinates` correcto; esto es, la lista de coordenadas en texto plano. Tan sólo nos falta convertir esa secuencia de coordenadas en una lista de objetos `GeoPoint`, tarea que llevamos a cabo mediante sencillas operaciones con la cadena:

```
String ruta = mCoordenadas.toString();
Log.d(this.TAG, "RUTA " + ruta);
String[] aRuta = ruta.trim().split(" ");
ArrayList<GeoPoint> listgp = new ArrayList<GeoPoint>();
GeoPoint gp;
for (int i = 0; i < aRuta.length; i++) {
      String[] coords = aRuta[i].split(",");
      try {
            gp = new GeoPoint((int) (Double.valueOf(coords[1]) * 1E6),
                        (int) (Double.valueOf(coords[0]) * 1E6));
            listgp.add(gp);
      } catch (Exception e) {
            Log.e(this.TAG,
                  "Error al transformar coordenadas: " + e.getMessage());
      }
}
return listgp;
```

Esta lista de objetos `GeoPoint` será recibida por la clase `EdificiosOverlay`, que a su vez se la enviará a `RutaOverlay` para que dibuje la ruta, y finalmente ambas capas serán incorporadas por `GUMapa` a la lista de *overlays* de la visual `MapView`, según todo lo que hemos visto en páginas anteriores.

Segunda Actividad: Plano de un edificio

La siguiente funcionalidad básica de la aplicación ideada originalmente, es la que permite mostrar al usuario un plano del edificio seleccionado. En principio, cualquier visor de imágenes podría llevar a cabo esta función, puesto que los planos de los edificios no dejan de ser imágenes normales y corrientes; de hecho, nuestra actividad podría llamar directamente al visor de imágenes nativo de Android y nuestros planos se mostrarían al usuario, permitiéndole incluso hacer zoom, sin ningún tipo de trabajo adicional.
Sin embargo, la visualización de las imágenes no es la única finalidad de nuestra nueva actividad. Si recordamos la descripción de la aplicación, a través del plano de cada edificio el usuario tendrá acceso a la lista de puntos de interés (POI) del edificio y podrá consultar información adicional de cada uno de ellos. Para lograr esto, nuestro visor particular tiene que ser capaz de mostrar sobre el plano la posición de cada uno de estos POI, y también tiene que poder detectar las acciones que el usuario lleve a cabo sobre los mismos (pulsaciones) para poder responder de manera adecuada a las mismas.

Figura 1. Actividad Plano, con el plano del edificio del antiguo Hospital de Marina cargado, y mostrando algunos puntos de interés situados en el mismo.

Por estos motivos se ha optado por crear una nueva actividad. Esta nueva actividad tendrá capacidades de desplazamiento y *zoom* de la imagen, igual que cualquier otro visor de imágenes; pero, adicionalmente, podrá ubicar sobre el plano los distintos POI asociados al mismo y, en caso de que el usuario pulse sobre alguno de ellos, mostrar información adicional e incluso abrir un navegador web para llevar al usuario a páginas relacionadas.

Visual XML

Como toda la funcionalidad adicional comentada se implementará a través del código del controlador, la visual XML de nuestra actividad es idéntica a la de cualquier otro visor básico de imágenes. Se muestra a continuación el contenido del fichero `visor.xml` que la define:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent" >
      <ImageView android:id="@+id/plano"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:src="@drawable/plano_defecto"
            android:scaleType="matrix" >
      </ImageView>
</FrameLayout>
```

El elemento `FrameLayout` define un contenedor de tipo marco, similar a los utilizados habitualmente en el desarrollo de aplicaciones gráficas con librerías como por ejemplo Swing; su contenido se mostrará a pantalla completa en el dispositivo del usuario. Dicho contenido, en nuestro caso, es un elemento `ImageView` que en tiempo de ejecución mostrará la imagen del plano solicitado por el usuario. Como única particularidad a destacar, tenemos el parámetro `scaleType="matrix"` que le estamos pasando al elemento `ImageView.` Este valor le indica a la visual que todas las operaciones de escalado de la imagen se calcularán en base a una matriz de coeficientes, gestionada por la clase principal y cuyo uso de la cual veremos más adelante.

Clase principal: GUPlano

Si en la actividad Mapa se utilizaban múltiples clases auxiliares para dar cabida a toda la funcionalidad del controlador –enfoque motivado por la necesidad de crear derivados específicos de `Overlay` y de implementar múltiples interfaces–, en la actividad Plano, por el contrario, toda la lógica se encuentra contenida dentro de la clase `GUPlano` y el flujo de la aplicación se controla mediante los eventos por ella recibidos. El único requisito para que la clase `GUPlano` pueda recibir estos eventos es que implemente la interfaz `OnTouchListener` (provista por la clase `View`), concretamente su método `onTouch(...)`. Posteriormente, durante la inicialización de la visual, nos presentamos a la misma como receptores de los eventos y, a partir de ese momento, cada vez que el usuario toque la pantalla de su dispositivo, el evento generado será procesado por la lógica de nuestro método `onTouch(...)`.

```
public class GUPlano extends Activity implements View.OnTouchListener {
	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.visor);
		ImageView view = (ImageView) findViewById(R.id.plano);
		view.setOnTouchListener(this);
```

Hemos visto que la visual XML asigna a la actividad un "plano por defecto", una imagen estándar que evita que se muestre la pantalla vacía. Durante la inicialización, es tarea del controlador sustituir esta imagen inicial por el plano correcto del edificio seleccionado por el usuario; para ello, hacemos uso del método `cambiarImagenPlano(ImageView)`:

```
private void cambiaImagenPlano(ImageView view) {
	// Cambiar imagen de la visual
	Edificio e = DatosAplicacion.getInstance()
				.getEdificio(mIndexEdificio);
	int imgID = getResources().getIdentifier(
				"plano_"+e.getImagenPlano()
				,"drawable", getPackageName());
	view.setImageDrawable(getResources().getDrawable(imgID));
	// Recalcular factor de escalado inicial
	DisplayMetrics metrics = new DisplayMetrics();
	getWindowManager().getDefaultDisplay().getMetrics(metrics);
	Options opt = new BitmapFactory.Options();
	opt.inDensity = metrics.densityDpi;
	opt.inTargetDensity = metrics.densityDpi;
	Bitmap bmp = BitmapFactory.decodeResource(
				getResources(), imgID, opt);
	float hOrig = bmp.getHeight();
	mEscalaInicial = view.getDrawable().getIntrinsicHeight() / hOrig;
}
```

Los primeros pasos que sigue este método se encargan de recuperar los datos del edificio seleccionado, entre los cuales se encuentra el nombre del recurso que contiene la imagen del plano. A partir de dicho nombre, se recupera el identificador global con el que el sistema operativo Android distingue este recurso, y con él se genera un objeto `Drawable` que se envía a la visual para que sustituya a su imagen por defecto.

Durante la fase de pruebas de la aplicación se detectó que, al cargar las imágenes mediante `Drawable`, Android les aplica un cierto escalado inicial para adaptar su tamaño al tamaño de pantalla del dispositivo. Esto implica que un punto concreto de la pantalla no tiene por qué corresponderse con el mismo punto sobre la imagen

original. Nuestra aplicación necesita por tanto conocer el factor de escalado que Android está aplicando en esta fase, ya que posteriormente tendrá que ser tenido en cuenta para obtener la correspondencia entre los eventos del usuario y los puntos de la pantalla. Los últimos pasos del método `cambiaImagenPlano(...)` se encargan de calcular este factor como el cociente entre las medidas del plano una vez cargado en la visual, y las medidas originales de un objeto `Bitmap` creado a partir de la misma imagen original.

Finalizado este método, disponemos ya del plano correcto en la visual y podemos continuar con el resto de inicializaciones, que se llevan a cabo en el método `onCreate(...)` de `GUPlano` y que se limitan a almacenar las medidas iniciales del plano para su posterior uso y a inicializar la matriz de transformaciones de la imagen. A partir de este momento, la actividad queda a la espera de recibir los eventos generados por las acciones del usuario, que explicaremos a continuación.

Gestión de eventos

Por implementar la interfaz `OnTouchListener`, nuestra clase dispone del método `onTouch(View, MotionEvent)`, encargado de recibir y gestionar los eventos que el usuario origina. Estos eventos están definidos en la clase MotionEvent y son los siguientes:

- `MotionEvent.ACTION_DOWN` es el evento generado cuando el usuario toca la pantalla con un dedo (o lápiz capacitivo o puntero similar). Incluye las coordenadas en las que se ha situado el dedo.
- `MotionEvent.ACTION_POINTER_DOWN` ocurre cuando el usuario sitúa un segundo dedo en la pantalla e incluye, además de las coordenadas del evento, un valor entero que identifica al nuevo puntero. Existen eventos adicionales para un mayor número de punteros, pero no es habitual su uso.
- `MotionEvent.ACTION_MOVE` se genera cada vez que el usuario mueve alguno de los dedos con los que está tocando la pantalla. Como información adicional, este evento incluye las coordenadas en las que se encuentra cada uno de los punteros cuando el movimiento finaliza. Nótese que no se incluyen las coordenadas de inicio del movimiento; es responsabilidad del programador almacenar esta información conforme la vaya recibiendo, por ejemplo cada vez que se recibe un `ACTION_DOWN.`
- `MotionEvent.ACTION_POINTER_UP` indica que el usuario ha levantado alguno de los dedos, pero no todos.
- `MotionEvent.ACTION_UP` se recibe cuando el usuario levanta el único dedo que seguía en contacto con la pantalla. Tanto este evento como el anterior vienen acompañados del identificador de puntero y de las últimas coordenadas en las que se situó.

Nuestro objetivo aquí es ser capaces de reconocer, identificando las secuencias de eventos generadas por el usuario, las tres acciones principales que puede llevar a cabo sobre la imagen: **desplazamiento** (***drag*** en inglés), que sucede cuando el

usuario desliza un dedo por la pantalla para mover la imagen; ***zoom*** o escalado, llevado a cabo cuando el usuario sitúa dos dedos sobre la pantalla y a continuación los separa o los acerca, para agrandar o empequeñecer la imagen respectivamente; y por último el habitual "toque" en la pantalla, equivalente al clic del ratón y que en inglés se conoce como ***tap***.

Evaluando los gestos que el usuario realiza sobre la pantalla para llevar a cabo cada una de estas acciones, podremos deducir el orden en que debemos tratarlos en el método `onTouch(...)` para que nuestra aplicación los reconozca y transforme la imagen (el plano del edificio) de manera acorde.

A continuación veremos con detalle cada una de las acciones y la lógica que siguen tanto a nivel de usuario como a nivel de código:

- **Desplazamiento *(drag)***: para desplazar la imagen por la pantalla, sin modificar su tamaño, el usuario realiza los siguientes gestos:

    1. Situar un dedo sobre la pantalla. Este gesto genera un evento de tipo `ACTION_DOWN`, que marca el inicio de la acción. Al recibir este evento, guardamos las coordenadas iniciales en la variable `start` y por otra parte guardamos también la matriz actual de transformaciones de la imagen (este elemento lo explicaremos más adelante).

    ```
    case MotionEvent.ACTION_DOWN:
          mode = DRAG;
          start.set(event.getX(), event.getY());
          savedMatrix.set(matrix);
          break;
    ```

    2. Desplazar el dedo para mover la imagen. Esto genera eventos de tipo `ACTION_MOVE`, de manera continua hasta que el usuario finalice la acción. Para cada uno de estos eventos, calculamos la diferencia respecto de las coordenadas origen de la acción (guardadas en el paso anterior) y la aplicamos sobre la matriz original como un desplazamiento.

```
public boolean onTouch(View v, MotionEvent event) {
      switch (event.getAction() & MotionEvent.ACTION_MASK) {
            case MotionEvent.ACTION_MOVE:
                  if (mode == DRAG) {
                        float dx = event.getX() - start.x;
                        float dy = event.getY() - start.y;
                        matrix.set(savedMatrix);
                        matrix.postTranslate(dx, dy);
                        break;
                  }
            }
            v.setImageMatrix(matrix);
            return true;
      }
```

Una vez finalizado el tratamiento del evento, aplicamos la matriz sobre la imagen para obtener la transformación deseada; incluimos en este caso la cabecera del método para mayor claridad.

3. Retirar el dedo. Este gesto genera un evento de tipo `ACTION_UP` que da por concluida la acción de desplazamiento. En este caso simplemente indicamos que ya no se está llevando a cabo ninguna acción.

```
case MotionEvent.ACTION_UP:
      mode = NONE;
      break;
```

Combinando los tres elementos anteriores dentro del método `onTouch(...)` y aplicando a la imagen del plano la matriz de transformaciones al final de cada evento, obtenemos el efecto deseado de desplazamiento continuo de la imagen cada vez que el usuario desliza un dedo sobre la pantalla. Sin embargo, tal cual está escrito, no existe un límite de desplazamiento de la imagen: el usuario podría mover la imagen hasta dejarla completamente fuera de pantalla. Para procurar que el plano del edificio sea siempre visible, se añade lógica adicional que controla que el desplazamiento final no supere los límites de la pantalla calculados durante la inicialización de la visual.

Esta parte del procesado del evento `ACTION_MOVE` obtiene de la matriz de transformaciones los valores actuales de desplazamiento y escalado, y los utiliza para precalcular las diferencias entre los límites del área visible de la pantalla, y los límites de la imagen una vez desplazada. Si alguno de los bordes de la imagen se sale del área visible, se modifica el valor de desplazamiento correspondiente `dx` o `dy` para forzarlo a que se mantenga dentro del área visible.

La lógica es la siguiente:

```
float dx = event.getX() - start.x;
float dy = event.getY() - start.y;

// limitar mover
matrix.getValues(matrixValues);
float currentY = matrixValues[Matrix.MTRANS_Y];
float currentX = matrixValues[Matrix.MTRANS_X];
float currentScale = matrixValues[Matrix.MSCALE_X];
float currentHeight = height * currentScale;
float currentWidth = width * currentScale;
float newX = currentX + dx;
float newY = currentY + dy;

RectF drawingRect = new RectF(newX, newY, newX + currentWidth,
                                    newY + currentHeight);
float diffUp = Math.min(viewRect.bottom - drawingRect.bottom,
                                viewRect.top - drawingRect.top);
float diffDown = Math.max(viewRect.bottom - drawingRect.bottom,
                                viewRect.top - drawingRect.top);
float diffLeft = Math.min(viewRect.left - drawingRect.left,
                                viewRect.right - drawingRect.right);
float diffRight = Math.max(viewRect.left - drawingRect.left,
                                viewRect.right - drawingRect.right);

if (diffUp + screenHeight > 0) { dy += diffUp + screenHeight; }
if (diffDown < 0) { dy += diffDown; }
if (diffLeft + screenWidth > 0) { dx += diffLeft + screenWidth; }
if (diffRight < 0) { dx += diffRight; }
```

- **Escalado (*zoom*)**: la secuencia de eventos para hacer *zoom* sobre el plano, utilizando dos dedos, es un poco más complicada. Sería la siguiente:

    1. Situar un dedo sobre la pantalla. Este gesto es el mismo con el que se inicia la acción de desplazamiento y, de hecho, en este punto todavía no sabemos cuál de las dos acciones tiene pensado llevar a cabo el usuario. La lógica, por tanto, sigue siendo la misma.

    2. Situar un segundo dedo sobre la pantalla. Este segundo gesto produce un nuevo evento, `ACTION_POINTER_DOWN`, que ahora sí nos asegura que el usuario pretende hacer *zoom* sobre el plano. En este caso los valores que necesitamos guardar son: la matriz que se está aplicando a la

imagen, para transformarla después; la distancia entre ambos dedos, pues de su variación dependerá que el escalado de la imagen; y el punto medio entre ambos dedos, que será el centro a partir del cual haremos que la imagen crezca o disminuya.

```
case MotionEvent.ACTION_POINTER_DOWN:
      oldDist = distancia(event);
      if (oldDist > 10f) {
            savedMatrix.set(matrix);
            fijaPuntoMedio(mid, event);
            mode = ZOOM;
      }
      break;
```

No obstante, imperfecciones en la superficie del dedo, la pantalla y la lógica de detección de eventos de Android, hacen que a veces, en las primeras versiones del sistema operativo, los eventos `ACTION_POINTER_DOWN` y `ACTION_DOWN` se produzcan al mismo tiempo cuando en realidad el usuario ha situado solo un dedo sobre la pantalla. Para evitar equívocos, añadimos una comprobación adicional que verifique que el evento se está originando en unas coordenadas distintas de las iniciales (las del primer dedo) y a una distancia mayor que cierto margen de seguridad.

3. Sin levantarlos de la pantalla, separar o acercar los dedos entre sí. Esto genera eventos `ACTION_MOVE` cuyas coordenadas utilizaremos para extraer la nueva distancia de separación de los dedos; calculando el ratio entre esta nueva distancia y la distancia anterior, obtendremos el factor de aumento que debemos aplicar sobre la imagen. Una vez calculado, aplicamos sobre la matriz de transformaciones el factor de escalado, a ambos ejes por igual para obtener un escalado uniforme del plano del edificio y centrándolo en el punto medio calculado al detectar la acción. Por último se aplicaría la matriz sobre la imagen mediante llamada al método `setImageMatrix(...)`, igual que en el caso anterior; omitimos el código para no sobrecargar el ejemplo.

```
case MotionEvent.ACTION_MOVE:
      if (mode == DRAG) {
            (...)
      } else if (mode == ZOOM) {
            float newDist = distancia(event);
            if (newDist > 10f) {
                  matrix.set(savedMatrix);
                  float scale = newDist / oldDist;
                  // limitar zoom
                  matrix.getValues(matrixValues);
                  float currentScale = matrixValues[Matrix.MSCALE_X];
                  if (scale * currentScale > maxZoom) {
                        scale = maxZoom / currentScale;
                  } else if (scale * currentScale < minZoom) {
                        scale = minZoom / currentScale;
                  }
                  matrix.postScale(scale, scale, mid.x, mid.y);
            }
      }
      break;
```

Antes de aplicar este escalado, se puede observar que llevamos a cabo una comprobación de máximos, precalculando el factor de *zoom* resultante y limitándolo a los valores predefinidos `maxZoom` o `minZoom` en caso necesario. Esto se hace así para evitar que el usuario pueda hacer *zoom* sin límite, ya que lo único que obtendría sería una imagen pixelada sin ningún tipo de utilidad salvo ralentizar el sistema debido al procesamiento necesario para obtenerla.

4. Levantar los dedos de la pantalla. Esto genera el evento `ACTION_POINTER_UP` para el primer dedo levantado y el evento `ACTION_UP` para el último dedo. En realidad, en lo que a la acción de escalado se refiere, basta con que el usuario levante uno de los dos dedos para considerar que la acción ha finalizado; por tanto, hacemos que el evento `ACTION_POINTER_UP` tenga la misma lógica de finalización ya existente, haciendo que su detección "caiga" hacia el evento `ACTION_UP` colocándolos en cascada.

```
case MotionEvent.ACTION_POINTER_UP:
case MotionEvent.ACTION_UP:
      mode = NONE;
      break;
```

Como hemos visto en el código, toda esta lógica se combina dentro del método

`onTouch(...)` y se utiliza la variable `mode` para diferenciar entre las diferentes acciones. Queda aún una acción por evaluar y es la acción equivalente al clic del ratón, el *tap.*

- **Pulsación (*tap*)**: esta acción es para el usuario la más sencilla de todas, pues consiste únicamente en tocar la pantalla con la punta del dedo en un punto determinado, como si fuera un botón; levantando posteriormente el dedo sin desplazarlo ni efectuar ninguna otra acción. El tratamiento de este evento sería como sigue:
  1. Poner un dedo en la pantalla. Este primer gesto es común a todas las acciones y no podemos utilizar el evento generado, `ACTION_DOWN`, para determinar si la pulsación corresponde a un *tap* o al inicio de otro movimiento. Se mantiene la misma lógica: guardar la matriz y coordenadas iniciales.
  2. Sin efectuar ningún movimiento adicional, retirar el dedo de la pantalla. La manera más sencilla de implementar esta condición es, al recibir el evento `ACTION_UP` que indica que se ha vuelto a retirar el dedo, comprobar si la posición del mismo es la misma que inicialmente. El problema es que el dedo humano, como puntero, es bastante impreciso debido a su fisionomía y, aunque el usuario no lo mueva, un simple cambio en la inclinación del dedo sobre la pantalla puede hacer que la coordenada detectada por el sistema operativo para el siguiente evento sea distinta de la inicial. Para facilitar la usabilidad de la aplicación es necesario incluir un pequeño margen de error en la comprobación, de por ejemplo diez píxeles, que dé cabida a pequeños movimientos involuntarios al llevar a cabo esta acción.

```
case MotionEvent.ACTION_POINTER_UP:
case MotionEvent.ACTION_UP:
      if (distancia(event, start) < 10f) {
            procesarTap(view, event);
      }
      mode = NONE;
      break;
```

El método `procesarTap(...)` es el encargado de gestionar esta acción en caso de que llegue a producirse. Su función es la de comprobar si el *tap* se ha llevado a cabo sobre un POI y, en tal caso, determinar cuál y lanzar el flujo previsto para esta acción. Lo veremos con más detalle posteriormente.

Matrices de transformación

Hemos visto que todas las transformaciones sobre la imagen se efectúan mediante la aplicación de una matriz de transformaciones, de acuerdo con el parámetro de configuración `scaleType="matrix"` especificado en la visual XML. La matriz es un objeto de la clase `Matrix` del paquete `android.graphics` y representa una matriz

matemática de tamaño 3x3, la cual es usada para almacenar los valores de perspectiva, escalado, sesgo y traslación de una imagen. Manipulando los valores de la matriz y aplicándola a una imagen, se pueden llevar a cabo operaciones como desplazamientos, rotaciones e inclinaciones de la imagen.
La potencia del uso de matrices radica en que se pueden encadenar varias transformaciones sobre la misma matriz mediante sucesivas operaciones. De esta forma podemos aplicar las mismas transformaciones a varias imágenes distintas, sin tener que repetir las operaciones cada vez, ni redibujar la imagen tras cada operación, ni perder ninguna de las transformaciones intermedias; y además podemos recuperar en cualquier momento los factores finales de transformación para aplicarlos sobre cualquier otro conjunto de puntos.
En nuestro caso, a la hora de precalcular las transformaciones sobre la imagen, hemos extraído de la matriz los factores de escalado y desplazamiento y los hemos aplicado manualmente sobre las coordenadas de la imagen para comparar el resultado con los límites prefijados. Lo mismo haremos más adelante para evaluar si un evento coincide con las coordenadas de un POI.
Cuando aplicamos matrices de transformación sobre la imagen, es el sistema operativo Android quien lleva a cabo todos estos cálculos para cada uno de los puntos que la componen, obteniendo la imagen transformada.

Tratamiento de los POI

Vimos que para procesar las pulsaciones del usuario sobre el plano, se había creado el método `procesarTap(...)` que comprobaba si las coordenadas en las que se había producido el evento se correspondían o no con un punto de interés marcado en el plano. Si bien es cierto que cada objeto Edificio lleva asociada una lista de objetos `EdificioPOI`, y cada `EdificioPOI` tiene definidas sus coordenadas respecto del plano del edificio... estas coordenadas no tienen por qué corresponderse con las recibidas por el evento *tap.* El motivo es que las coordenadas que vienen informadas en el objeto `MotionEvent` se corresponden con la pantalla del dispositivo, no con la imagen mostrada. Por eso, si estamos efectuando una acción *tap* sobre un punto de una imagen que ha sido aumentada y desplazada, el evento no tendrá las mismas coordenadas que ese mismo punto en la imagen original.
Para sortear este obstáculo, el método debe traducir las coordenadas primero. El código de este método es el siguiente:

```
private void procesarTap(ImageView view, MotionEvent event) {
      float[] puntosOriginales = { event.getX(), event.getY() };
      savedMatrix.getValues(matrixValues);
      puntosOriginales[0] -= matrixValues[Matrix.MTRANS_X];
      puntosOriginales[1] -= matrixValues[Matrix.MTRANS_Y];
      puntosOriginales[0] /= matrixValues[Matrix.MSCALE_X] *
mEscalaInicial;
      puntosOriginales[1] /= matrixValues[Matrix.MSCALE_X] *
mEscalaInicial;
      int xOrig = (int) puntosOriginales[0];
      int yOrig = (int) puntosOriginales[1];
      EdificioPOI poi = buscaPOI(xOrig, yOrig);
      if (poi != null) {
            Toast.makeText(this, poi.getNombre(), Toast.LENGTH_LONG);
            mostrarInfoPOI(poi);
      }
}
```

El método recupera de la matriz de transformación los factores de desplazamiento y escalado actuales, y los aplica a la inversa sobre las coordenadas del evento. De esta forma se deshacen todos los cambios realizados sobre las coordenadas originales de la imagen, y se obtienen unos puntos que ya se pueden comparar con aquellos de la lista de POI.

Precisamente esto es lo que hace el método `buscarPOI(...)`:

```
private EdificioPOI buscaPOI(int x, int y) {
      Edificio e = DatosAplicacion.getInstance()
                        .getEdificio(mIndexEdificio);
      EdificioPOI poi = null;
      for (EdificioPOI p : e.getListaPOI()) {
            if (distancia(p.getX(), p.getY(), x, y) < 20f) {
                  poi = p;
                  break;
            }
      }
      return poi;
}
```

La lógica es sencilla: recorre la lista de POI asociada al edificio actual, y compara las coordenadas de cada uno de ellos con las coordenadas del evento (una vez deshechas las transformaciones). Si encuentra algún POI en un radio de 20 píxeles alrededor del punto origen del evento, finaliza la búsqueda y lo devuelve. Si no encuentra ninguno en dicho radio, el método devuelve `null`.

El método padre `procesarTap(...)` comprueba si el objeto EdificioPOI recibido es nulo o no. En caso de que no lo sea, quiere decir que el usuario ha pulsado sobre un POI del plano para obtener información del mismo; de satisfacer esta demanda es de lo que se encarga el método `mostrarInfoPOI(...)`:

```
private void mostrarInfoPOI(final EdificioPOI poi) {
	AlertDialog.Builder dialog = new AlertDialog.Builder(this);
	dialog.setTitle(poi.getNombre());
	dialog.setMessage(poi.getDescripcion());
	dialog.setPositiveButton("Más info", new OnClickListener() {
		public void onClick(DialogInterface dialog, int which) {
			dialog.dismiss();
			String url = poi.getURL();
			startActivity(new Intent(new Intent(
					Intent.ACTION_VIEW, Uri.parse(url))));
		}
	});
	dialog.setNegativeButton("Atrás", new OnClickListener() {
		public void onClick(DialogInterface dialog, int which) {
			dialog.dismiss();
		}
	});
	dialog.show();
}
```

El método construye y muestra un diálogo con el nombre y la descripción del POI recibido, contenidos en el objeto `EdificioPOI.` Este diálogo, además de ofrecer la información básica del POI, permite también acceder a la información ampliada vía web: pulsando el botón "Más info" se abrirá en el navegador predeterminado la página web asociada al POI. El botón "Atrás" descarta la información y permite al usuario regresar al plano del edificio.

Con esto finaliza la explicación de la actividad Plano. Como decíamos al principio, es una actividad en la que toda la lógica de la misma se encuentra contenida dentro de una única clase, la principal. Es un enfoque distinto motivado por tener un único método de gestión de eventos disponible así como por no necesitar de la utilización de API ni librerías externas que obliguen a implementar otros métodos.

La situación actual del desarrollo nos brinda una aplicación que ya dispone de las funcionalidades completas de Mapa y Plano de edificios. Este era el concepto original de la aplicación: inicialmente no existía ningún otro requisito. Sin embargo, una aplicación así se queda corta y en la siguiente sección veremos el porqué.

Consulta de un edificio concreto: Listado

Una vez implementadas las funcionalidades básicas de mapa de edificios y plano de edificio, se mostró la aplicación en esta fase "alfa" a un grupo escogido de usuarios para que la probara.
Durante estas pruebas se observó que algunos de ellos, a la hora de consultar un edificio concreto, se veían obligados a ir pulsando aleatoriamente los edificios del mapa hasta encontrar el que buscaban, puesto que el mapa muestra la posición de cada edificio únicamente mediante un icono, y los usuarios que no estaban familiarizados con el campus de la UPCT no conseguían identificar cada edificio sólo por su posición en el mapa.

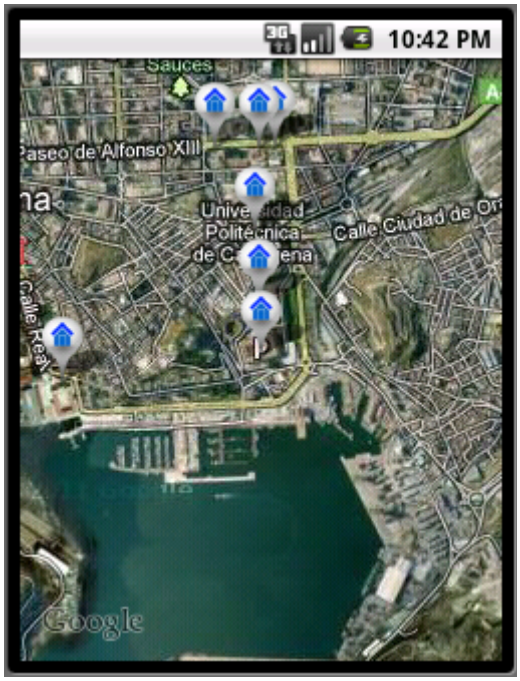

Figura 1: mapa de edificios.

Estos usuarios resaltaron dicha dificultad de uso y propusieron como solución una nueva pantalla, distinta del mapa, que permitiera buscar un edificio por el nombre y mostrarlo seleccionado en el mapa.

La actividad Listado satisface esta demanda del usuario, incorporando a la aplicación una nueva pantalla que muestra una lista de edificios. Para cada elemento de la lista, es decir para cada edificio, se muestra una pequeña foto, el nombre del edificio, y el nombre de la escuela u órgano que está asociada al edificio.

Al pulsar uno de estos elementos, la aplicación inicialmente mostraba el mapa de edificios centrado en el edificio seleccionado, facilitando así al usuario la localización del mismo. No obstante, la evolución posterior de la aplicación ha modificado este

comportamiento y ahora, al pulsar en cualquiera de los edificios del listado, se muestra la pantalla Ficha con los datos de ese edificio en concreto.

Figura 2: listado de edificios.

Para añadir la funcionalidad de listado se han creado cuatro nuevos elementos: la clase `GUListado`, que implementa la lógica de control de la actividad y de los eventos; la clase `EdificioAdapter`, para facilitar la creación de la lista y que veremos más adelante; y las visuales XML `lista.xml` y `edificio_lista.xml`.

Visuales XML

Las visuales XML son muy sencillas. Por un lado tenemos el XML de la lista propiamente dicha, `lista.xml`, cuyo único contenido es un elemento `ListView` (que representa la lista), y un elemento `TextView` auxiliar que mostrará en pantalla un texto en caso de que la lista no contenga ningún item.

La clase `ListView` es una visual compuesta, esto es, una visual que puede contener otras visuales. La visual principal del elemento `ListView` provee un marco con capacidades de *scroll* (la lista), y mediante una clase adaptador se van creando e insertando automáticamente en la lista los elementos que la componen.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" >
      <ListView
          android:id="@+id/android:list"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          />
      <TextView
          android:id="@+id/android:empty"
          android:text="@string/lbListaVacia"/>
</LinearLayout>
```

Los elementos de la lista, por otro lado, tienen su propia visual, `edificio_lista.xml`. Esta visual es más clásica y se compone de tres elementos: un elemento `ImageView`, que en tiempo de ejecución mostrará una imagen del edificio correspondiente a esa posición de la lista; y dos elementos `TextView` que contendrán el nombre del edificio y el nombre de la escuela asociada. La distribución de estos campos se consigue mediante elementos `LinearLayout` con orientación horizontal o vertical:


```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="?android:attr/listPreferredItemHeight"
    android:padding="6dip">
    <ImageView
        android:id="@+id/imgLista"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_marginRight="6dip"
        android:src="@drawable/ficha_defecto" />
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="0dip"
        android:layout_weight="1"
        android:layout_height="fill_parent">
        <TextView
            android:id="@+id/lbEntidad"
            android:layout_width="fill_parent"
            android:layout_height="0dip"
            android:layout_weight="1"
            android:gravity="center_vertical"
        />
        <TextView
            android:id="@+id/lbEdificio"
            android:layout_width="fill_parent"
            android:layout_height="0dip"
            android:layout_weight="1"
            android:singleLine="true"
            android:ellipsize="marquee"
        />
    </LinearLayout>
</LinearLayout>
```

Figura 3: relación entre la visual en pantalla y en XML de un item de la lista de edificios.

Clase principal: GUListado

La clase que implementa la actividad, `GUListado`, es un tipo especial de actividad que hereda de la clase `ListActivity`. Este tipo especial de actividad facilita la creación y el control de visuales de tipo lista, y gracias a ello la clase `GUListado` es también muy sencilla; sus funciones se reducen a inicializar la visual de la actividad y a recibir los eventos desde la misma.
Para inicializar la visual, el método `onCreate(...)` sigue tres pasos:

- seleccionar el XML `lista.xml` como visual de la actividad;
- crear un objeto de la clase que se vaya a utilizar como adaptador, en nuestro caso `EdificioAdapter`;
- pasárselo a la lista mediante el método `setListAdapter(ListAdapter)`.

```
/**
 * Punto de entrada de la actividad. Gestiona toda la lógica de control.
 * @see android.app.Activity#onCreate(android.os.Bundle)
 */
@Override
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.lista);
      this.mAdapter = new EdificioAdapter(this, R.layout.edificio_lista,
                  DatosAplicacion.getInstance().getListaEdificios());
      setListAdapter(this.mAdapter);
}
```

Hecho esto, la visual junto con el adaptador se encargan de cargar y mostrar la lista y sus elementos, y de enviar las acciones sobre cada item (clic, clic prolongado, etc.) de vuelta a la actividad para su tratamiento. Este tratamiento de eventos se lleva a cabo mediante los métodos provistos por la clase `ListActivity`, los cuales hereda nuestra clase `GUListado`.
Para los eventos de tipo clic, los únicos que nos interesan, tenemos el método `onListItemClick(...)`. Cada vez que se pulsa un elemento de la lista, este método recibe la información del mismo, entre la cual se encuentra el identificador (posición dentro de la lista) del edificio seleccionado. Nuestra intención es que se muestre el edificio seleccionado, ya sea en el mapa o en la ficha; por tanto, el método no tiene más que crear el objeto `Intent` adecuado (para `GUMapa` o para `GUFicha`) y pasarle como datos extra el identificador del edificio.

```
/**
 * Evento de clic sobre un elemento de la lista.
 * Crea un <code>Intent</code> de la Ficha con el id
 * del edificio seleccionado y lo ejecuta.
 */
@Override
protected void onListItemClick(ListView l, View v, int position,
long id) {
      super.onListItemClick(l, v, position, id);
      Intent i = new Intent(this, GUFicha.class);
      i.putExtra(GUFicha.KEY_EDIFICIO, position);
      startActivity(i);
}
```

Una vez lanzada la actividad, depende de ella extraer los datos del edificio; nuestra actividad queda en segundo plano hasta que finalice la actividad lanzada o se cierre la aplicación.

Clase auxiliar: EdificioAdapter

Por último tenemos la clase adaptador. Los adaptadores son clases utilizadas como puente entre la visual de los items de una lista, y el origen de los datos que éstos representan. Un adaptador recibe como parámetros el XML de la visual del elemento y el origen de los datos, y va creando instancias de la visual que inicializa con los datos obtenidos.
La API de Android ofrece varios adaptadores distintos, cada uno pensado para un origen de datos diferente: existen adaptadores basados en un array de elementos, adaptadores que utilizan un cursor de base de datos para recuperar la información, e incluso adaptadores para crear los item a partir de otras listas. Todos ellos tienen en común que implementan, directa o indirectamente, la interfaz `ListAdapter`.

En nuestro caso, la información de los edificios está contenida estáticamente en un `ArrayList`, con lo cual un `ArrayAdapter` podría servirnos. Sin embargo, el `ArrayAdapter` por defecto sólo está preparado para asociar cada elemento del array a un objeto de tipo `TextView`; esta funcionalidad es demasiado limitada para los elementos de nuestra lista, que deben contener una imagen y dos textos distintos (nombre de edificio y nombre de entidad), y por tanto nos vemos obligados a extenderla mediante una clase nueva: `EdificioAdapter`.

```
/**
 * Constructor.
 * @param context contexto de la aplicación
 * @param textViewResourceId id del recurso de tipo layout
 * (visual) que se quiere obtener
 * @param items lista de edificios
 */
public EdificioAdapter(Context context, int textViewResourceId,
            ArrayList<Edificio> items) {
      super(context, textViewResourceId, items);
      this.items = items;
      this.mLayout = textViewResourceId;
}
```

`EdificioAdapter` recibe de `GUListado`, al ser construido, tres parámetros: una referencia a la actividad (contexto), una referencia a la visual XML que debe ser utilizada para cada elemento de la lista, y una referencia al `ArrayList` común que contiene la información de los edificios en objetos `Edificio`.

La clase `EdificioAdapter` conoce internamente la correspondencia entre cada uno de los campos de la clase `Edificio` y el elemento correspondiente de la visual `edificio_lista.xml`, y con esta información el método `EdificioAdapter.getView(...)` se encarga posteriormente de acceder a cada una de las posiciones del array, extraer la información de cada edificio, instanciar la visual a partir del XML y rellenar cada campo con el dato correcto, para finalmente retornar a su llamante el objeto visual creado. La lógica de este método se puede consultar en el apéndice correspondiente.

Cabe destacar que esta arquitectura supone un alto acoplamiento entre las capas del modelo MVC; no obstante tiene la ventaja de que simplifica enormemente tanto la definición de la visual XML, como la lógica contenida en la propia actividad; es por ello que se ha considerado adecuada para su uso en la aplicación.

Dando opciones al usuario: actividad Inicio

Conforme se introducen nuevas actividades en la aplicación, como es el caso de la creación de la actividad Listado como alternativa a la actividad Mapa, aumentan también las tareas que hay que llevar a cabo como inicialización previa. Por ejemplo, la actividad Mapa podría querer comprobar si existe acceso a internet antes de ejecutarse, o quizá la actividad Listado tenga que descargar las imágenes de los edificios de algún servidor.

Estas tareas se tienen que llevar a cabo antes de mostrar la información al usuario; sin embargo, no es en absoluto recomendable que el usuario se tenga que quedar "esperando" a que la aplicación arranque, sin ningún tipo de señal que indique que efectivamente la aplicación se está ejecutando correctamente.

Muchos programas solucionan esta situación mediante el uso de una imagen o pantalla inicial, denominada *splash screen* en inglés. Una *splash screen* es esa pequeña imagen de bienvenida, a veces con una barra o unos mensajes de progreso (o ambos), que muestran algunas aplicaciones al arrancar mientras llevan a cabo en segundo plano sus rutinas de inicialización.

Figura 1. Pantalla de entrada o splash screen *mostrada durante el arranque de la aplicación Eclipse (entorno de desarrollo utilizado en el proyecto).*

Mediante el uso de una pantalla como esta, se hace saber al usuario que la aplicación está arrancando, aún cuando la funcionalidad no esté disponible todavía porque se estén llevando a cabo rutinas de inicialización necesarias para su posterior uso.
Con este mismo fin se plantea la creación de una splash screen para nuestra aplicación, que se mostraría al usuario al inicio mientras se cargan los datos y se comprueban las conexiones.

Por otro lado, ahora que la aplicación dispone de varias funcionalidades distintas, se cree conveniente también ofrecerle al usuario alguna forma de elegir entre las mismas. Dado que la actividad Mapa frustraba la experiencia de uso de algunos usuarios, parece adecuado ofrecer a dichos usuarios la opción de acceder directamente al listado de edificios, sin pasar por el mapa. De igual forma, los usuarios que ya conocen la situación de cada edificio pueden aprovechar mejor el tiempo accediendo directamente al mapa sin pasar por el listado.

Tenemos, por tanto, dos nuevas necesidades: implementación de una pantalla inicial que ocupe al usuario mientras se llevan a cabo las tareas de inicialización, e implementación de algún mecanismo de selección o alternancia entre las distintas actividades de la aplicación.

Algunas de las opciones que se han considerado para implementar la solución a ambas necesidades (menú de actividades y pantalla de espera) han sido las siguientes:

- Menú nativo de Android:
  El sistema operativo Android incluye de forma nativa la capacidad de mostrar un menú sencillo, el cual aparece al pulsar la tecla MENU del dispositivo. Este menú permitiría al usuario cambiar de una actividad a otra.
  - Ventajas:
    - Práctica estándar: la utilización de menús para presentarle al usuario las acciones disponibles en una actividad, es una práctica estándar en las aplicaciones Android. Por ello, su implementación está perfectamente documentada, y su uso resulta familiar para quien tenga experiencia previa con Android.
  - Inconvenientes:
    - Diseño: el diseño de los menús de Android es ajeno al diseño del resto de la aplicación, que utiliza diálogos con botones para las acciones de usuario.
    - Efecto no inmediato: dado que por defecto el menú está oculto, habría que notificar al usuario la existencia del mismo de alguna forma; de lo contrario puede darse el caso de que el usuario (por desconocimiento o falta de familiaridad con el dispositivo) nunca descubra que existe la opción de consultar el listado de edificios y no obtenga por tanto una experiencia plena de la aplicación.
    - Solución parcial: esta opción no resuelve la necesidad de una pantalla de espera inicial o *splash screen*, que tendría que ser implementada por separado.

- Pantalla de inicio:
  Se puede añadir una actividad adicional como primera pantalla de la aplicación, la cual presentaría al usuario las opciones disponibles (mapa y listado, en un principio).

- Ventajas:
  - Diseño: al ser una actividad completa, el diseño está totalmente controlado por nosotros y por tanto podemos adecuarlo al del resto de pantallas de la aplicación. Además da a la aplicación un aspecto más depurado, al dotarla de una pantalla de inicio con su logo y título correspondientes.
  - Efecto inmediato: el usuario conocerá desde el primer momento las opciones de que dispone, pues será la primera pantalla que vea al arrancar la aplicación.
  - Independiente del dispositivo: esta opción no depende de la existencia de una tecla MENU que muestre el menú; se basa única y exclusivamente en los clics que el usuario hace con el dedo, de igual forma que en el resto de la aplicación.
  - *Splash screen*: al crear una pantalla de inicio completa, eliminamos la necesidad de una pantalla adicional de espera.
- Inconvenientes:
  - Elimina el menú: dado que la práctica estándar en Android es incluir las opciones en un menú, a algunos usuarios puede resultarles extraño que la aplicación no tenga menú. Por lo demás, el uso de una pantalla de inicio no parece representar ningún problema ni funcional ni de usabilidad.

Figura 1. Pantalla de inicio de la aplicación, en su versión final. Incluye el botón Ayuda, actividad que se explicará en el siguiente capítulo.

Una vez evaluadas las ventajas e inconvenientes de cada una, la opción lógica es la creación de una pantalla de inicio. Al lanzar la aplicación, la primera pantalla que verá el usuario será una pantalla de bienvenida, que mostrará el logo de la aplicación y los botones de acceso a las funciones principales.

Configuración como actividad principal

Si queremos que esta pantalla sea la primera actividad de nuestra aplicación, debemos definirla como tal en el fichero de configuración de la aplicación Android, `AndroidManifest.xml`. Se consigue de la siguiente manera:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
            package="org.proyecto.guiaupct"
            ...>
    <application
            ...
            <uses-library android:name="com.google.android.maps" />
        <activity android:name=".GUInicio">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".mapa.GUMapa" />
        <activity android:name=".mapa.GUPlano" />
        <activity android:name=".listado.GUListado" />
    </application>
...
```

Visuales XML

La visual de la pantalla de inicio es muy simple. Presenta, en disposición vertical:

- una imagen que representa el logotipo de la aplicación, ocupando la mitad superior de la pantalla
- el título de la aplicación, justo debajo del logotipo
- los controles de selección de actividad en la parte inferior de la visual.

Estos controles, a su vez, no son más que botones dispuestos de manera horizontal, cada cual con el nombre de su actividad asociada.

A continuación se muestra el contenido del fichero `splash.xml` que define esta visual.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:background="#75B2DE" >
      <ImageView android:id="@+id/splashscreen"
            ...
            android:src="@drawable/splash"
            android:layout_gravity="center" />
      <TextView android:id="@+id/tvTitulo"
            ...
            android:layout_weight="1"
            android:gravity="center"
            android:text="@string/lbTitulo"
            android:textSize="18sp"
            android:textStyle="bold"
            android:textColor="#FFFFFF" />
      <LinearLayout
            android:orientation="horizontal"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="center" >
            <Button android:id="@+id/btMapa"
                  android:text="@string/lbMapa"
                  android:layout_weight="1"
                  ... />
            <Button android:id="@+id/btLista"
                  android:text="@string/lbLista"
                  android:layout_weight="1"
                  ... />
            <Button android:id="@+id/btAyuda"
                  android:text="@string/lbAyuda"
                  android:layout_weight="1"
                  ... />
      </LinearLayout>
</LinearLayout>
```

Como punto a destacar, la visual fija un color de fondo en tono azul, similar al color de fondo de la imagen utilizada como logotipo. Se consigue así un efecto de transparencia del logotipo que, al ser redondo, se vería enmarcado en un cuadrado azul de no utilizarse ese mismo color como fondo.

Actividad principal

La clase principal de la actividad, `GUInicio`, se limita prácticamente a recuperar las instancias de los botones definidos por la visual XML y asignar al evento de pulsación

de cada uno la lógica de lanzamiento de la actividad asociada a dicho botón.

SI bien el objetivo inicial de esta pantalla era informar al usuario mientras se llevaban a cabo ciertas comprobaciones de inicio, en la versión final se han eliminado estas tareas. No obstante, dado que la pantalla sigue cumpliendo su función de menú, se ha mantenido, dejando abierta la posibilidad de que futuras expansiones hagan uso de la misma para las tareas de inicio.

```
@Override
public void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.splash);

	Button btMapa = (Button) findViewById(R.id.btMapa);
	Button btLista = (Button) findViewById(R.id.btLista);
	Button btBuscar = (Button) findViewById(R.id.btBuscar);

	btMapa.setOnClickListener(new View.OnClickListener() {
		public void onClick(View view) {
			startActivity(new Intent(GUInicio.this, GUMapa.class));
		}
	});
	btLista.setOnClickListener(new View.OnClickListener() {
		public void onClick(View view) {
			startActivity(new Intent(GUInicio.this,
GUListado.class));
		}
	});
	btBuscar.setOnClickListener(new View.OnClickListener() {
		public void onClick(View view) {
			startActivity(new Intent(GUInicio.this,
GUAyuda.class));
		}
	});
}
```

Existe sin embargo una última tarea que esta clase también lleva a cabo, y es la de finalización de la aplicación: cuando el usuario, siguiendo el flujo de navegación natural, regrese a esta pantalla y pulse nuevamente la tecla ATRÁS, se indicará al sistema operativo que la aplicación ha terminado y que puede eliminarse de la pila de procesos.

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
      if (keyCode == KeyEvent.KEYCODE_BACK) {
            this.finish();
            return true;
      }
      return super.onKeyDown(keyCode, event);
}
```

Actividad: Ayuda

La pantalla de ayuda es la actividad más sencilla de toda la aplicación GUÍA UPCT, pues su único cometido es mostrar al usuario, en formato texto, una descripción de la aplicación, sus objetivos, requisitos y limitaciones, así como una explicación de la funcionalidad de cada una de las pantallas.

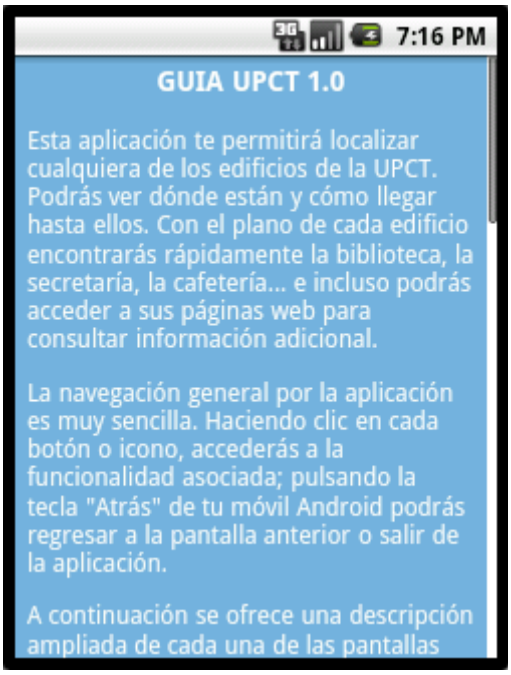

Figura 1. Ayuda de la aplicación "GUÍA UPCT".

Inicialmente, se planteó incluir el texto de la ayuda como parte de los recursos de tipo "cadena de texto" de la aplicación; esto es, dividirlo en secciones y almacenarlo en el fichero `strings.xml` incluido en el paquete de recursos de cualquier aplicación Android. La composición de la ayuda a partir de estos elementos unitarios, se realizaría a posteriori en la visual XML, utilizando las características disponibles para maquetar la ayuda en secciones o similar. No obstante, esta opción no fue la elegida finalmente, por motivos que veremos a continuación.

Red WiFi UPCT

Una de las necesidades que tiene la aplicación para acceder a toda su funcionalidad, es la de conexión a internet para la utilización de las funcionalidades de Mapa y de acceso a páginas web online, tanto de las escuelas (accesibles a través de su ficha) como de los puntos de interés (accesibles a través del plano del edificio correspondiente). Dado que el público objetivo de la aplicación son sobre todo estudiantes de primer y segundo año —los cuales normalmente tienen unos recursos económicos limitados—, es posible que un alumno que esté utilizando la aplicación

no tenga contratada una tarifa de datos y, por tanto, incurra en un gasto cada vez que haga uso de la conectividad 3G de su dispositivo. Esta situación debe evitarse en lo posible: no es aceptable que una aplicación orientada al alumno, genere al mismo un perjuicio económico derivado de su uso, no ya por la mala publicidad que esto supondría para la aplicación (que también), sino por principio personal del autor.
Afortunadamente, la UPCT pone a disposición de todos sus alumnos y visitantes una red WiFi de acceso libre y gratuito, la cual puede ser utilizada por el alumno usuario de la aplicación para acceder sin trabas a toda la funcionalidad existente. De esta manera se evita parcialmente que los alumnos dependan de su propia conexión de datos para utilizar la aplicación y se mitiga la problemática anteriormente descrita.

Ayuda en formato HTML

Para informar al usuario de la necesidad de conexión de datos y la posibilidad de utilizar la red WiFi de la Universidad, se consideró adecuado incluir como parte de la ayuda una nota informativa al respecto. Dado que la configuración de la red WiFi de la UPCT y los requisitos de acceso a la misma vienen explicados en la web de la Universidad, se optó por incluir en la ayuda un enlace externo a dicha web. Es este último punto el que entra en conflicto con el concepto inicial de la ayuda generada mediante cadenas de texto. En principio, aunque una visual por defecto puede recuperar el texto de la ayuda, no hay una manera clara de transformar dicho texto en un enlace de hipertexto que permita al usuario acceder a la información alojada en la página web de la Universidad. Se pensó en incluir, a modo de enlace, una imagen sobre la cual poder detectar pulsaciones y lanzar la actividad adecuada (en este caso el navegador web). Sin embargo, una segunda revisión de este concepto llevó a la conclusión de que sería mucho más fácil componer el texto de la ayuda utilizando el lenguaje HTML. De esta forma, todo el contenido de la ayuda se almacenaría en un único fichero de tipo HTML, lo cual simplifica posibles actualizaciones o modificaciones de la ayuda; además, la utilización del lenguaje HTML permite incluir elementos útiles como imágenes, listas o enlaces externos, de una manera más directa y de sobra conocida por cualquier desarrollador.

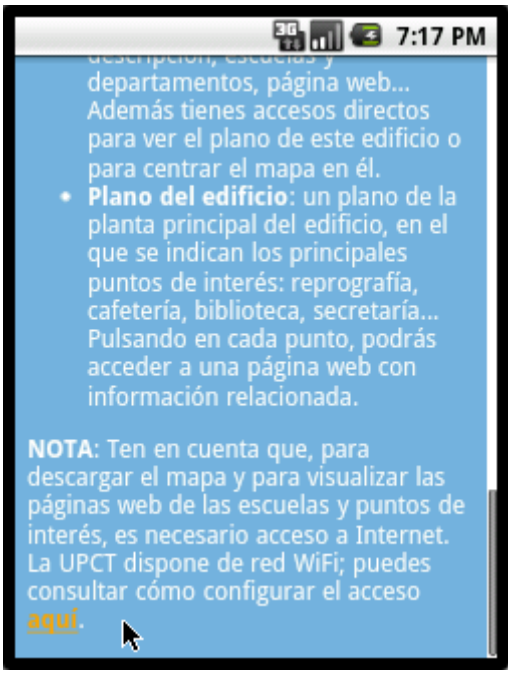

Figura 2. Pantalla de ayuda, mostrando la nota final con el enlace HTML a la información sobre configuración de acceso a la red WiFi de la UPCT.

Visuales XML

Para poder presentar al usuario un texto en formato HTML, tan sólo hace falta utilizar una visual de tipo `WebView`. Este tipo de visual utiliza un motor basado en WebKit para presentar el contenido de recursos HTML, sean locales o remotos, con el formato adecuado y además gestiona automáticamente los eventos de pulsación de enlaces externos, lanzando el navegador web del dispositivo en caso necesario sin necesidad de añadir código adicional a la actividad. El fichero `ayuda.xml` refleja la sencillez de uso de esta visual:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView  xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@+id/webview"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
/>
```

Actividad principal

La clase `GUAyuda` es la que contiene la lógica principal de la actividad, y es prácticamente igual de sencilla que la visual XML. Se limita a instanciar el objeto `WebView` según está definido en la visual XML, y a configurar algunos aspectos del

mismo. Por último le pasa el recurso HTML que debe mostrar por defecto. A partir de ese momento es el objeto `WebView` el que gestiona las funcionalidades típicas de navegación web.

```
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.ayuda);

      mWebView = (WebView) findViewById(R.id.webview);
      mWebView.getSettings().setJavaScriptEnabled(true);
      mWebView.loadUrl("file:///android_asset/ayuda.html");
}
```

Contenido de la ayuda

Para terminar, mostramos a continuación el contenido del fichero HTML que contiene la ayuda de la aplicación. Como elementos destacables podemos observar, por ejemplo, el bloque de reglas CSS que se ha utilizado para aplicar a cada elemento de la página algunos colores y decoraciones. El lenguaje CSS (*Cascaded Style Sheet*) permite una definición más clara de los estilos de una página web y es un estándar en la programación web. En este caso se ha utilizado para dotar a la página web de la misma combinación de fondo azul y texto blanco que tiene la pantalla de inicio de la aplicación, unificando estilos; para resaltar los hiperenlaces, en cambio, se ha escogido el color naranja que representa a nuestra ingeniería :-)
También encontramos que se ha incluido, tal como se comentaba antes, una imagen, concretamente el icono utilizado por la actividad Mapa para representar a los edificios, que aparece en la ayuda como parte de la explicación que se le ofrece al usuario sobre esa pantalla; y al final de todo podemos ver el enlace externo a la web de la Universidad, donde se explica cómo configurar el acceso a la red WiFi.

```
<html>
   <head>
      <title>Ayuda GUÍA UPCT</title>
      <style type="text/css">
         body { background-color:#75B2DE; }
         p { color:white; }
         p.titulo { color:white; font-size:large; text-align:center; }
         ul { color:white; }
         a { color:orange; font-weight:bold; }
      </style>
   </head>
   <body>
      <p class="titulo">
      <b>GUIA UPCT 1.0</b>
      </p>
      <p>
       Esta aplicación te permitirá localizar cualquiera de los edificios de la UPCT. Podrás ver
dónde están y cómo llegar hasta ellos. Con el plano de cada edificio encontrarás rápidamente la
biblioteca, la secretaría, la cafetería... e incluso podrás acceder a sus páginas web para
consultar información adicional.
      </p>
      <p>
      La navegación general por la aplicación es muy sencilla. Haciendo clic en cada botón o icono,
accederás a la funcionalidad asociada; pulsando la tecla "Atrás" de tu móvil Android podrás
regresar a la pantalla anterior o salir de la aplicación.
      </p>
      <p>
       A continuación se ofrece una descripción ampliada de cada una de las pantallas que componen
la aplicación:
      <ul>
            <li><b>Pantalla de inicio</b>: en esta pantalla podrás acceder a las dos funciones
principales de la aplicación (mapa y listado), así como a esta pantalla de ayuda. Pulsando la tecla
"Atrás" en tu móvil desde esta pantalla, abandonarás la aplicación.
         <li><b>Explorar Mapa</b>: gracias a la API de Google Maps, aquí podrás saber de un vistazo
tu posición actual y la localización de cada edificio de la Universidad, marcados con el icono <img
src="marcador.png"></img>. Si pulsas en dicho icono, podrás ver directamente la ruta para llegar al
mismo, o acceder a la ficha de ese edificio.
          <li><b>Lista de edificios</b>: desde aquí podrás ver un listado de los edificios de la
Universidad, mostrando para cada uno de ellos una pequeña imagen, el nombre del edificio y el
nombre de la(s) escuela(s) que alberga. Pulsando en cada elemento de la lista, accederás a la ficha
de cada edificio.
          <li><b>Ficha del edificio</b>: en esta pantalla podrás consultar toda la información del
edificio: foto, descripción, escuelas y departamentos, página web... Además tienes accesos directos
para ver el plano de este edificio o para centrar el mapa en él.
         <li><b>Plano del edificio</b>: un plano de la planta principal del edificio, en el que se
indican los principales puntos de interés: reprografía, cafetería, biblioteca, secretaría...
Pulsando en cada punto, podrás acceder a una página web con información relacionada.
      </ul>
      </p>
      <p>
       <b>NOTA</b>: Ten en cuenta que, para descargar el mapa y para visualizar las páginas web de
las escuelas y puntos de interés, es necesario acceso a Internet. La UPCT dispone de red WiFi;
puedes consultar cómo configurar el acceso <a
href="http://www.upct.es/~si/serv_al/wifi.php">aquí</a>.
      </p>
   </body>
</html>
```

Unificando la información: DatosAplicacion

Hasta ahora no hemos entrado en detalle sobre el origen de los datos que utiliza nuestra aplicación. A lo largo de los apartados anteriores hemos estado hablando de edificios, coordenadas, nombres y descripciones, listas de puntos de interés... sin explicar de dónde estamos recuperando dichos datos. En esta sección nos ocuparemos de estos temas y de las distintas decisiones tomadas al respecto a lo largo del desarrollo.

Estructura de los datos

Ya hemos visto, durante la explicación de lo que sería el Modelo de nuestra aplicación, que los datos de edificios y puntos de interés se iban a almacenar en objetos de tipo `Edificio` y `EdificioPOI`, las clases creadas para este papel, pero como es normal tendremos que agrupar estos objetos en alguna estructura que facilite su acceso. En la aplicación se van a mostrar principalmente listas de edificios, ya sea en el Mapa o en el Listado. Una estructura de tipo enumeración o conjunto podría servir a nuestro propósito, pero también tenemos actividades como la Ficha, la cual veremos más adelante, que requieren poder acceder a un `Edificio` en concreto a través de un índice. Lo mismo aplica para los listados de puntos de interés.

En ambos casos, una buena elección es utilizar una estructura de tipo `ArrayList`, parametrizada para objetos de tipo `Edificio` o `EdificioPOI`, según corresponda. Esta estructura dinámica combina las propiedades de una lista y de un *array*, y permite así recorrer sus datos con los métodos propios de una lista (primero, último, siguiente) pero también acceder directamente al elemento situado en una posición concreta a través de su índice. Al ser dinámica, además, no tenemos que definir un tamaño inicial del *array* o lista, sino que automáticamente se reserva o libera el espacio necesario cada vez que se añade o elimina un objeto de la colección.

La inicialización de los datos mediante estas estructuras, de una forma estática, sería algo así:

```
private void inicializaEdificios() {
      // Lista de edificios
      mEdificios = new ArrayList<Edificio>();
      // Añadimos cada uno de los edificios en objetos Edificio
      mEdificios.add(new Edificio("Antiguo Hospital de Marina",
                                  "E.T.S. Ing. Industrial",
                                  "Descripción del edificio",
                                  "http://www.industriales.upct.es",
                                  new GeoPoint(37599549,-978919),
                                  "muralla", "muralla"));
      mEdificios.add(new Edificio("Cuartel de Antigones",
                                  "E.T.S. Ing. Telecomunicaciones",
...));
      mEdificios.add(new Edificio(...));
      // Añadimos a cada edificio sus puntos de interés
      mEdificios.get(0).addPOI(new EdificioPOI(59, 70,
                                               "Aulas de Informática",
...));
      mEdificios.get(0).addPOI(new EdificioPOI(...));
      mEdificios.get(1).addPOI(new EdificioPOI(...));
}
```

En los inicios del proyecto, las únicas actividades que estaba previsto desarrollar eran el Mapa y el Plano. En esa etapa de desarrollo, la actividad Mapa no enviaba al Plano el objeto `Edificio`, sino que le pasaba directamente la información necesaria (ruta de la imagen) para visualizar el plano correcto en cada caso; debido a esto, era más fácil y rápido a nivel de desarrollo incluir los datos de los edificios de manera estática en la propia actividad Mapa, en la forma descrita arriba.

Centralización de los datos

En el momento en el que añadimos nuevas actividades a la aplicación, en cambio, aparece la necesidad de que varias clases tengan acceso a los mismos datos sobre los edificios. Para conseguirlo, podríamos simplemente copiar las estructuras de datos de una actividad a otra, es cierto; al fin y al cabo, sobre los mismos no se hace ningún tipo de modificación que tenga que ser compartida por el resto de actividades, con lo cual bastaría con que cada actividad tuviera una copia de las listas utilizadas y las accediera de manera independiente. De hecho, en un principio y por los mismos motivos, esto es exactamente lo que se hizo en la actividad Listado: una copia local de las listas de edificios.

Pero conforme se incrementa el número de actividades, y por tanto de copias, este enfoque deja de ser práctico. Recordemos que los datos son estáticos: están incrustados dentro del propio código fuente. Al tener varias copias de los mismos repartidas por el código, cada cambio que tengamos que hacer, cada nuevo edificio que queramos añadir o descripción de un POI que queramos alterar nos obligan a

modificar todas las copias para mantenerlas sincronizadas, forzando una recompilación de clases que no tendrían por qué verse afectadas por este tipo de actuaciones sobre los datos de la aplicación. Además, a nivel de rendimiento también nos vemos afectados: la duplicación de las estructuras a nivel de código tiene su contrapartida en el aumento del uso de memoria RAM, un recurso muy preciado en los dispositivos móviles y que estamos desaprovechando cada vez que una actividad carga datos que, en realidad, ya han sido cargados previamente por otra actividad de la misma aplicación.

Por el contrario, si unificamos todos los datos en una única clase que sirva de punto de acceso para todas las actividades, obtendremos una serie de ventajas:

- **Rendimiento**: reducción de la cantidad total de memoria utilizada por la aplicación, al evitar la duplicidad innecesaria de los datos y de las estructuras que los contienen.

- **Comodidad**: agrupación de todos los datos en un único punto, facilitando la modificación de los datos de los edificios y POI en el caso de mantenerse el caracter estático de los mismos.

- **Modularidad**: si en un futuro se decidiera aplicar alguna de las mejoras propuestas para la aplicación, como es la de hacer que los datos se obtengan de un XML configurable o de manera dinámica (via web o similar), la implementación sería mucho más sencilla al afectar únicamente a la clase que contiene los datos comunes, siempre y cuando se mantenga la interfaz con el resto de actividades.

Es por este motivo que surge la clase `DatosAplicacion.` Su objetivo es unificar todos los datos de edificios y puntos de interés, evitando duplicidades, aumentando la eficiencia de los accesos a los datos por parte del resto de actividades, y proporcionando un punto de acceso único que facilite la implantación de nuevas validaciones sobre el acceso a los datos o de carga dinámica de los mismos.

Para implementar la clase `DatosAplicacion` se plantean dos alternativas. Una de ellas es implementarla como una subclase de la clase `Application`, una clase que en Android se utiliza cuando varias actividades necesitan compartir un estado global. Ésta es una solución adaptada específicamente a las características de Android, a la estructura y ciclo de vida de sus aplicaciones. Sin embargo, precisamente por ser específico de Android, no es un buen ejemplo de aplicación de patrones conocidos.

La otra opción es implementarla como una clase independiente utilizando el patrón *Singleton*, aprovechando las características que éste proporciona. Por su valor didáctico hemos escogido esta última opción, ya que el escenario presentado: una clase que tiene que ofrecer a otras un punto de acceso único sobre unos datos que no

queremos que se dupliquen en memoria, conforma una oportunidad ideal para mostrar la utilidad del patrón y cómo utilizarlo. Como ya hemos explicado en la sección correspondiente las características del patrón *Singleton* y su implementación genérica, ahora veremos cómo se trasladan estas especificaciones a la clase `DatosAplicacion`.

Implementación del patrón *Singleton*

La clase define un miembro estático privado de tipo `DatosAplicacion` que, una vez inicializado, será la única instancia de la clase, compartida por todas las otras clases de la aplicación. Además, esta instancia se inicializa en la misma declaración, para evitar ciertos problemas de sincronismo derivados del uso de inicialización *lazy:*

```
/**
 * Instancia única de la clase.
 * Se inicializa en la definición para evitar problemas con múltiples
hilos.
 */
private static DatosAplicacion singleton = new DatosAplicacion();
```

Puesto que el *singleton* es privado, la única forma de que otra clase obtenga la referencia a la instancia del mismo será a través de un método propio `getInstance()`, también estático, pero público:

```
/**
 * Devuelve la única instancia de la aplicación.
 * @return el objeto <i>singleton</i> DatosAplicacion.
 */
public static DatosAplicacion getInstance() {
      return singleton;
}
```

El constructor de la clase se define también como privado:

```
/**
 * Constructor privado para evitar instancias no deseadas.
 */
private DatosAplicacion() {
      inicializaEdificios();
}
```

De esta forma, además de evitar que otras clases fabriquen instancias de la nuestra, se bloquea el mecanismo de herencia, impidiendo que se puedan crear subclases

herederas de `DatosAplicacion` que duplicarían las referencias al *singleton*.

Dentro del constructor es donde se lleva a cabo la inicialización de la lista de edificios, mediante el método privado estático `inicializaEdificios()`. El método va añadiendo a un `ArrayList`, de manera similar a como hemos visto al principio, objetos `Edificio` y `EdificioPOI` con la información necesaria. Este `ArrayList`, a su vez, está declarado en la clase también como variable privada estática a la cual sólo se puede acceder mediante los métodos públicos definidos para ese fin:

```
/**
 * Lista de objetos <code>Edificio</code> con la información de los
edificios.
 */
private static ArrayList<Edificio> mEdificios;

/**
 * @return la lista de edificios con la información de los mismos.
 */
public ArrayList<Edificio> getListaEdificios() {
      return DatosAplicacion.mEdificios;
}

/**
 * Devuelve un edificio concreto a través de su índice en la lista,
 * siendo 0 el índice del primer edificio.
 * @param id índice del edificio que se quiere obtener
 * @return objeto <code>Edificio</code> que ocupa la
 * posición <code>id</code> de la lista.
 */
public Edificio getEdificio(int id) {
      return mEdificios.get(id);
}
```

Así conseguimos que todos los datos estén centralizados en la misma clase, que en memoria sólo tendrá una instancia para ahorrar recursos y asegurar que todas las actividades presentan los mismos datos, y que facilitará modificaciones de los mismos o de la lógica para obtenerlos.

¿Y si no hay conexión? Actividad: Ficha

Con las pantallas principales de nuestra aplicación ya codificadas y prácticamente terminadas, durante las pruebas empezó a hacerse evidente un pequeño *handicap* de nuestra aplicación, derivado del uso de servicios externos para obtener parte de la funcionalidad: la dependencia de la conexión de datos.

Como sabemos, la actividad Mapa utiliza los servicios y API de Google Maps tanto para mostrarnos el mapa de edificios, como para calcular las rutas seleccionadas. Para utilizar dichos servicios, es imprescindible que el dispositivo Android disponga de una conexión a Internet activa; sin una conexión de datos, no resulta posible descargar las imágenes satelitales y por tanto el mapa se convierte en una pantalla en blanco sin usabilidad ninguna. Esta pérdida de funcionalidad, que ya de por sí afecta bastante a nuestra aplicación, tiene además un efecto colateral: perdemos también el acceso a la actividad Plano ¡aunque ésta no use la conexión de datos para nada! Así es: dado que al Plano de cada edificio sólo se puede acceder (de momento) seleccionándolo primero en el Mapa, si no podemos cargar correctamente el Mapa entonces la actividad Plano queda fuera del alcance de las acciones del usuario, limitadas por la ausencia de conexión.

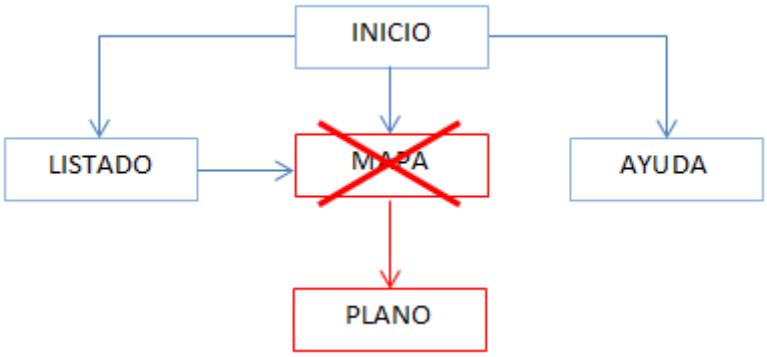

Figura 1. Pérdida de acceso a la actividad Plano en un escenario de bloqueo de la actividad Mapa por falta de una conexión de datos.

Ante este escenario, nos planteamos como posible solución hacer una comprobación previa (aprovechando la actividad Inicio, según vimos en la sección dedicada a ella) para confirmar que disponemos de conexión a Internet activa, y deshabilitar el acceso al Mapa en caso contrario; pero ¿qué sentido tendría nuestra aplicación entonces? Si desactivamos el Mapa (y por tanto el Plano) en ausencia de una conexión de datos, los usuarios que se encuentren en esa situación sólo podrían ejecutar las actividades Listado y Ayuda: una experiencia pobre en comparación con lo que nuestra GUÍA UPCT puede ofrecer, y que provocaría, con toda seguridad, que el usuario desinstale la aplicación al no poder darle un uso aceptable cuando está en modo *offline*.

Necesidad de funcionalidad *offline*

Se concluye de lo anterior que el enfoque "mapa-centrista" que hemos adoptado en el desarrollo de la aplicación hasta este punto, quizá no es el más adecuado si queremos ofrecer una aplicación útil al mayor número de usuarios posible, incluidos aquellos que no tienen en su dispositivo conexión a Internet. Es cierto que el concepto original de la aplicación versaba sobre un mapa que nos permitiera guiarnos por el campus de la Universidad, pero ya hemos visto que esta visión nos crea una dependencia de la conexión de datos que no es deseable.

¿Cuál puede ser la alternativa, entonces? Si no hay conexión, al Mapa no merece la pena acceder, eso está claro; pero ¿y al Plano? Realmente, para acceder a la imagen del plano del edificio no necesitamos conexión, y de hecho ni siquiera necesitamos el Mapa: la información de las imágenes está guardada en cada uno de los objetos `Edificio`. ¿Y qué otra actividad, aparte del Mapa, es la que nos muestra los edificios disponibles? Exacto: el Listado. A través de la actividad Listado, podemos seleccionar cualquier edificio de la aplicación sin necesidad de una conexión de datos, y utilizar su información para que el usuario pueda al menos consultar el plano del edificio seleccionado, aunque no pueda ver su situación en el mapa.

En este punto del desarrollo, al seleccionar un edificio de la lista, la actividad Listado llama a la actividad Mapa pasándole el identificador del edificio, para que se muestre el mapa centrado en las coordenadas de dicho edificio. Podríamos simplemente modificar esta llamada para que la actividad ejecutada fuera el Plano en vez del Mapa. No debemos olvidar, no obstante, que el sentido de incluir la actividad Listado era para facilitar a los usuarios la localización en el Mapa de aquellos edificios que sólo conocían por su nombre o su escuela; si eliminamos esta funcionalidad, sólo estaremos sustituyendo un problema por otro.

Tenemos entonces que incluir un nuevo diálogo o menú que, al pulsar en cualquiera de los edificios del Listado, le dé al usuario la opción de navegar al Mapa o al Plano, posibilitando así el acceso a este último aún cuando el usuario no tenga conexión a Internet. Eso está bien, pero ¿por qué detenernos ahí? Puestos a incluir nuevos elementos, podemos ir más allá: podemos crear una actividad completa, una pantalla nueva que, además de facilitar la elección anterior, muestre al usuario toda la información disponible *offline* sobre el edificio, con mejores fotografías, una descripción más detallada...

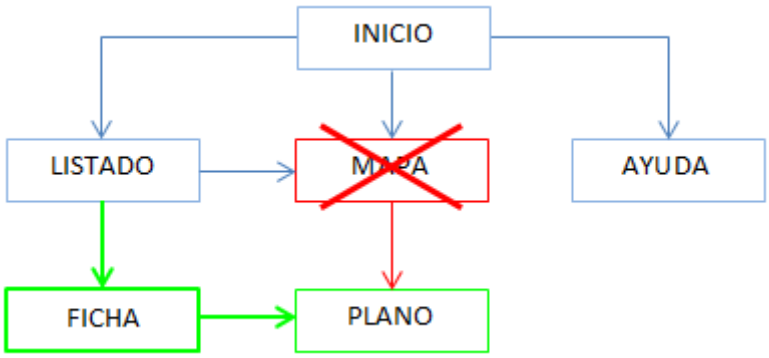

Figura 2. La inclusión de la actividad Ficha recupera el acceso a la actividad Plano y ofrece funcionalidad adicional aún cuando no se disponga de conexión de datos.

Así es como surge la actividad Ficha: una actividad que muestra al usuario una ficha completa del edificio seleccionado: nombre, fotografía, escuela, descripción de su historia... Junto con controles que permiten acceder desde la actividad Ficha a las actividades Mapa y Plano, e incluso a la página web de la entidad que alberga el edificio.

Con la inclusión de esta nueva actividad, el concepto de la aplicación GUÍA UPCT se desplaza de su enfoque inicial "mapa-centrista" a un nuevo enfoque "edificio-centrista", según el cual el flujo principal de la aplicación se centra más en la información de cada edificio concreto y en sus datos y puntos de interés, y el Mapa se convierte en una actividad accesoria que nos facilita la llegada a los mismos. Además, puesto que la información de los edificios se encuentra almacenada en la propia aplicación –independientemente de que posibles mejoras alteren la forma de obtener esta información, aunque se hiciera dinámicamente a través de Internet, una vez obtenida tendría que almacenarse igualmente para su utilización–, este cambio de enfoque aumenta sobremanera el valor y la utilidad que nuestra aplicación ofrece al usuario, el cual podrá seguir utilizando la GUÍA UPCT como una guía clásica de edificios, aún cuando la ausencia de conexión no le permita usar la funcionalidad de orientación mediante el mapa.

Figura 3. Ficha del edificio del antiguo Cuartel de Antigones, sede de la Escuela Técnica Superior de Ingeniería de Telecomunicaciones. Puede observarse en la barra de información que el dispositivo tiene las conexiones de datos y voz deshabilitadas.

Visual XML

La visual XML de la Ficha es, si no la más elaborada (honor que le corresponde a la del Listado, compuesta de dos ficheros), sí la más extensa, debido a que engloba en un único fichero XML una gran cantidad de elementos distintos para poder mostrar toda la información y controles necesarios. Por este motivo, vamos a exponer los elementos en dos bloques, a fin de que se vea más claramente la estructura.

El primer bloque persigue únicamente proveer a la Ficha de un marco que permita la funcionalidad de desplazamiento o *scroll* vertical. Por defecto, las visuales de las actividades Android no incorporan esta funcionalidad; si queremos incluirla debemos añadir a nuestra visual XML un elemento `ScrollView` que englobe cualesquiera otros elementos que creamos que, por su extensión, podrían salirse del área visible de la ventana. En nuestro caso concreto, uno de estos elementos podría ser por ejemplo la descripción del edificio: si pretendemos que la Ficha ofrezca datos sobre la historia del edificio, año de construcción, usos anteriores... es probable que la longitud del texto de la descripción vaya más allá de los límites de la ventana y que por tanto sea necesaria la aplicación del elemento `ScrollView` para permitir al usuario visualizar todo el contenido de la pantalla. Para ello, definimos un `ScrollView` dentro de un `LinearLayout` vertical, para aplicarlo a esa dimensión, y dentro del `ScrollView` definimos lo que sería el `LinearLayout` base del resto de componentes, también con orientación vertical que es la que seguirán los elementos visibles de la Ficha.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent" >
      <ScrollView
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" >
            <LinearLayout
                  android:orientation="vertical"
                  android:layout_width="fill_parent"
                  android:layout_height="fill_parent" >
                  <... resto de elementos ...>
            </LinearLayout>
      </ScrollView>
</LinearLayout>
```

Los elementos visibles de la Ficha, por su parte, se definen con elementos TextView e ImageView ya conocidos, siguiendo en sentido vertical de arriba a abajo el siguiente orden:

- Título: nombre del edificio
- Fotografía del edificio
- Nombre de la escuela u órgano de gobierno
- Controles de navegación
- Descripción del edificio

```
<TextView
      android:id="@+id/lbEdificio"
      android:layout_height="0dip"
      android:layout_weight="1"
      android:gravity="center_vertical"
      android:singleLine="true"
      android:ellipsize="marquee" />
<ImageView
      android:id="@+id/imgFicha"
      android:layout_height="wrap_content"
      android:layout_marginBottom="6dip"
      android:src="@drawable/ficha_defecto" />
<TextView
      android:id="@+id/lbEntidad"
      android:layout_height="0dip"
      android:layout_weight="1"
      android:gravity="center_vertical" />
<LinearLayout android:orientation="horizontal">
      ...
</LinearLayout>
<TextView
      android:id="@+id/lbDescripcion"
      android:layout_height="0dip"
      android:layout_weight="1"
      android:gravity="center_vertical" />
```

Para el título de la Ficha se han añadido dos configuraciones especiales: `singleLine="true"` y `ellipsize="marquee"`. En el caso de que el edificio tuviera un nombre muy largo, a través de estas configuraciones la visual aplicaría al título un efecto de marquesina, permitiendo que se lea en su totalidad. El resto de propiedades son las habituales de orientación, tamaño, etc. ya comentadas en anteriores apartados.

Por último tenemos los controles de navegación, una fila de botones que aparecen en horizontal dentro de la disposición vertical general:

```
<LinearLayout
      android:orientation="horizontal"
      android:layout_height="wrap_content"
      android:layout_gravity="center" >
      <Button android:id="@+id/btFichaMapa"
            android:text="@string/lbFichaMapa"
            android:layout_weight="1"
            android:layout_height="40dip" />
      <Button android:id="@+id/btFichaPlano"
            android:text="@string/lbFichaPlano"
            android:layout_weight="1"
            android:layout_height="40dip" />
      <Button android:id="@+id/btFichaWeb"
            android:text="@string/lbFichaWeb"
            android:layout_weight="1"
            android:layout_height="40dip" />
</LinearLayout>
```

Esto se consigue, tal como hicimos para los elementos del Listado, incluyendo un `LinearLayout` horizontal como un elemento más, justo antes de la descripción.

Clase principal: GUFicha

Si hubiéramos diseñado la clase `GUFicha` en una etapa más temprana del desarrollo, la estructura lógica de la misma habría sido algo parecido:

1. Recuperar el identificador del `Edificio` que queremos mostrar en la Ficha.
2. Construir una instancia Java de la visual XML y asignarla como nuestra `View` principal.
3. Rellenar cada uno de los campos de la ficha con la información correspondiente del objeto `Edificio`, y asignar la imagen adecuada al objeto `ImageView`.
4. Asociar a los controles de navegación, eventos que lancen la actividad solicitada cuando el usuario los pulse.

La lógica habría sido sencilla y directa, digna de convertirse en ejemplo en un tutorial cualquiera sobre cómo iniciarse en la programación Android, pero por lo demás no habría tenido especial valor didáctico.

Por suerte para nosotros, a estas alturas del desarrollo, esta secuencia de pasos (al menos los 3 primeros) nos debería resultar familiar, por el simple hecho de que ya hemos implementado otra clase que hace exactamente lo mismo: `EdificioAdapter`. Por este motivo, hemos pensado que sería más interesante desde el punto de vista técnico hacer un ejercicio de reutilización de esta clase, adaptándola para que pueda ser usada tanto por la actividad Listado como por la actividad Ficha. Los detalles de la reutilización se explicarán en detalle más adelante; de momento nos vamos a seguir

centrando en la clase GUFicha y en su lógica particular.

Lo primero que hace la clase es, como ya hemos indicado, recuperar la información de estado y escoger la visual ficha.xml como View principal.

```
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.ficha);
```

A continuación, igual que se hacía en la clase GUListado, la clase GUFicha crea una instancia del adaptador EdificioAdapter y la inicializa pasándole la visual escogida y la lista de objetos Edificio que recupera de nuestro *singleton*, DatosAplicacion.

```
final DatosAplicacion instance = DatosAplicacion.getInstance();
this.mAdapter = new EdificioAdapter(this, R.layout.ficha,
                                    instance.getListaEdificios());
```

A partir de aquí es donde se diferencian las implementaciones de GUListado y GUFicha para hacer uso del adaptador. La clase GUListado, al utilizar una visual de tipo ListView, incorporaba de serie toda la lógica necesaria para ir pasándole al objeto EdificioAdapter los índices adecuados y que éste le devolviera en cada ocasión una nueva View ya rellena con la información del Edificio correcto. La clase GUFicha, por el contrario, no dispone de tal funcionalidad y tenemos que ser nosotros los que obtengamos el índice correcto y solicitemos a EdificioAdapter que nos genere la View con los datos adecuados. De eso es de lo que se encargan las siguientes líneas de código:

```
if (savedInstanceState != null) {
      mIndexEdificio = savedInstanceState.getInt(KEY_EDIFICIO);
      setContentView(mAdapter.getView(mIndexEdificio, null, null));
} else if (getIntent().getExtras() != null) {
      mIndexEdificio = getIntent().getExtras().getInt(KEY_EDIFICIO);
      setContentView(mAdapter.getView(mIndexEdificio, null, null));
}
```

La Ficha no es una actividad que el usuario pueda lanzar de manera directa; siempre va a ser ejecutada desde otra actividad precedente, la cual será la encargada de indicarle el edificio que debemos mostrar. Así, el valor de la variable mIndexEdificio se obtendrá, o bien de los datos de estado guardados en una ejecución anterior, si es que existen (primera condición), o bien de los datos

adicionales que nos haya enviado alguna otra Actividad a través del `Intent` que simboliza la llamada.

En cualquiera de los dos casos, este índice se pasa como parámetro al método `getView(...)` del adaptador, el cual retorna una instancia de la `View` configurada inicialmente (`ficha.xml`) rellena con los datos del `Edificio` que en el `ArrayList` ocupe esa posición.

Seguidamente se recuperan de la visual ya construida, las instancias de los botones de navegación, para poder asignarles acciones a los eventos correspondientes.

```
Button btFichaMapa = (Button) findViewById(R.id.btFichaMapa);
Button btFichaPlano = (Button) findViewById(R.id.btFichaPlano);
Button btFichaWeb = (Button) findViewById(R.id.btFichaWeb);
```

Cada uno de ellos creará, cuando sea pulsado, un objeto `Intent` del tipo adecuado, que podrá referirse a una actividad de la aplicación (caso de los botones para el Mapa y el Plano) o a una acción genérica del sistema, como es `ACTION_VIEW`, para visualizar la página web del edificio en el navegador por defecto.

```
btFichaMapa.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
            Intent i = new Intent(getApplicationContext(),
                        GUMapa.class);
            i.putExtra(GUMapa.KEY_CENTRAR, true);
            i.putExtra(GUMapa.KEY_LAT,
instance.getEdificio(mIndexEdificio)
                                    .getCoordenadas().getLatitudeE6());
            i.putExtra(GUMapa.KEY_LON,
instance.getEdificio(mIndexEdificio)
                                    .getCoordenadas().getLongitudeE6());
            startActivity(i);
      }
});

btFichaPlano.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
            Intent i = new Intent(GUFicha.this, GUPlano.class);
            i.putExtra(GUPlano.KEY_EDIFICIO, mIndexEdificio);
            GUFicha.this.startActivity(i);
      }
});

btFichaWeb.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
            String url =
instance.getEdificio(mIndexEdificio).getPaginaWeb();
            startActivity(new Intent(Intent.ACTION_VIEW,
Uri.parse(url)));
      }
});
```

En el botón "Ver mapa", la información que se envía a la actividad `GUMapa` junto con el Intent son los datos de latitud y longitud, no el índice del `Edificio` seleccionado. Esto es así porque, en las fases iniciales del proyecto, estos eran los datos básicos que utilizaba la actividad Mapa para centrarse sobre un edificio, y cuando se implementó la Ficha se consideró que no merecía la pena cambiar la lógica del Mapa, dejándola así por motivos históricos, por así decirlo. Aún y así, tanto en este caso como en el del botón "Ver página web" los datos que se envían a las actividades se obtienen directamente desde el *singleton* `DatosAplicacion` utilizando el índice obtenido durante la inicialización de la actividad Ficha. En el caso del botón "Ver plano", la clase `GUFicha` envía a la clase `GUPlano` el índice del `Edificio` tal cual, sin añadir ningún otro dato, ya que será la propia actividad Plano la que recupere la información del plano y los POI directamente del *singleton* a través de este índice.

Reutilización de EdificioAdapter

Si recordamos, la clase `EdificioAdapter` se utilizaba en la clase `GUListado` para instanciar y rellenar automáticamente las vistas de cada uno de los elementos de la lista. Su utilización estaba justificada por el hecho de que los elementos no eran simples cadenas de texto (que habrían sido gestionadas automáticamente por la lista), sino visuales compuestas de imágenes y texto.
En la sección dedicada a la actividad Listado, explicamos cómo la actividad principal obtenía de `EdificioAdapter` los objetos `View` ya inicializados, mediante llamadas al método `getView(...)`. Lo que no mencionamos entonces es que la implementación del método `getView(...)` también la hicimos nosotros: es tarea del programador indicarle a `EdificioAdapter` la correspondencia entre los elementos de la visual XML y los campos del objeto base, dado que es totalmente arbitraria.

La implementación inicial del método era así:

```
public View getView(int position, View convertView, ViewGroup parent) {
      View v = convertView;
      if (v == null) {
            LayoutInflater vi = (LayoutInflater) getContext()
                        .getSystemService(Context.LAYOUT_INFLATER_SERVIC
E);
            v = vi.inflate(mLayout, null);
      }
      Edificio e = items.get(position);
      if (e != null) {
            ImageView imgLista = (ImageView)
v.findViewById(R.id.imgLista);
            TextView entidad = (TextView)
v.findViewById(R.id.lbEntidad);
            TextView edificio = (TextView)
v.findViewById(R.id.lbEdificio);
            int imgID = getContext().getResources().getIdentifier(
                        "lista_" + e.getImagenFoto(), "drawable",
                        getContext().getPackageName());
            imgLista.setImageDrawable(getContext().getResources()
                        .getDrawable(imgID));
            entidad.setText(e.getEntidad());
            edificio.setText(e.getNombre());
      }
      return v;
}
```

Como se puede observar, se obtiene del `ArrayList` el objeto `Edificio` referenciado; de la visual `v` configurada (`edificio_lista`), cada uno de sus elementos; y se inicializa cada uno de estos en base a aquél. Las líneas iniciales son para generar una

nueva instancia del objeto `View` en caso de que el llamante no nos proporcione una.

Precisamente por haberla implementado nosotros, resulta muy sencillo adaptar la lógica del método `getView(...)` para incluir los elementos que la visual XML de la Ficha no comparte con las visuales del Listado. Basta con recuperar las referencias a los nuevos campos y rellenarlas con la información adecuada, igual que los ya existentes.

Hay que tener cuidado, sin embargo: dado que ahora el objeto EdificioAdapter puede estar utilizando una cualquiera de dos visuales distintas, debemos incluir algo de lógica de comprobación de errores, para verificar que los campos que estamos intentando rellenar, efectivamente se encuentran disponibles. Por ejemplo, si estamos intentando rellenar una Ficha, el campo imgLista será null. De igual forma, si estamos rellenando el Listado, el campo descripción no existe en la visual lista.xml y por tanto será null en tiempo de ejecución.

El método quedaría finalmente como se muestra a continuación. Se han resaltado las líneas de la nueva lógica para facilitar la comparación entre el antes y el después:

```
public View getView(int position, View convertView, ViewGroup parent) {
    View v = convertView;
    if (v == null) {
        LayoutInflater vi = (LayoutInflater) getContext()
            .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        v = vi.inflate(mLayout, null);
    }
    Edificio e = items.get(position);
    if (e != null) {
        ImageView imgLista = (ImageView) v.findViewById(R.id.imgLista);
        ImageView imgFicha = (ImageView) v.findViewById(R.id.imgFicha);
        TextView entidad = (TextView) v.findViewById(R.id.lbEntidad);
        TextView edificio = (TextView) v.findViewById(R.id.lbEdificio);
        TextView descripcion = (TextView)
v.findViewById(R.id.lbDescripcion);
        if (imgLista != null) {
            int imgID = getContext().getResources().getIdentifier(
                    "lista_" + e.getImagenFoto(), "drawable",
                    getContext().getPackageName());
            imgLista.setImageDrawable(getContext().getResources()
                    .getDrawable(imgID));
        }
        if (imgFicha != null) {
            int imgID = getContext().getResources().getIdentifier(
                    "ficha_" + e.getImagenFoto(), "drawable",
                    getContext().getPackageName());
            imgFicha.setImageDrawable(getContext().getResources()
                    .getDrawable(imgID));
        }
        if (entidad != null) { entidad.setText(e.getEntidad()); }
        if (edificio != null) { edificio.setText(e.getNombre()); }
        if (descripcion != null)
{ descripcion.setText(e.getDescripcion()); }
    }
    return v;
}
```

De esta manera podemos hacer que el mismo adaptador que nos genera las visuales de cada uno de los elementos del Listado de edificios, nos genere también las visuales de cada una de las Fichas de edificio – siempre y cuando le proporcionemos el identificador adecuado (cosa que `GUListado` hacía automáticamente).

Esto tiene una consecuencia curiosa: si en este punto sustituimos en la clase `GUListado` "`eficio_lista`" por "`ficha`", obtendríamos efectivamente, sin ninguna otra modificación, una lista de edificios en la que cada elemento sería una Ficha completa. Una prueba más de la versatilidad del mecanismo de visuales XML.

Interacción entre actividades y ciclo de vida

Uno de los aspectos que todavía no hemos visto en detalle, es el paso de información entre actividades. Hemos mencionado que las actividades se envían entre sí el identificador del edificio seleccionado –o, en el caso de las actividades que se comunican con la actividad Mapa, las coordenadas del mismo–, pero hemos dejado la explicación para el final por ser un mecanismo sencillo y común a todas las actividades.

Cuando se ejecuta, una actividad puede obtener sus datos iniciales de dos maneras:

- A través de información extra enviada por otra actividad junto con el objeto `Intent`.
- A través de la información de estado guardada en el objeto `savedInstanceState` por la última instancia que se ejecutó de la actividad.

Paso de datos entre actividades

Para llamar a otra actividad, como ya hemos visto, nuestras clases tienen que crear objetos `Intent` del tipo adecuado para que el sistema operativo localice la actividad asociada y la ejecute. Por ejemplo, si quisiéramos ejecutar la actividad Plano, bastaría con estas dos líneas de código:

```
Intent i = new Intent(getApplicationContext(), GUPlano.class);
startActivity(i);
```

Al llamar a `startActivity(Intent)` con un objeto `Intent` de la clase `GUPlano`, le estamos indicando al sistema operativo que nuestra intención es ejecutar la actividad asociada a esa clase, si es que existe. Todo esto ya lo conocemos, pero ¿cómo le indicamos a la actividad Plano el plano que tiene que mostrar? Tenemos que hacerle llegar, junto con la llamada, algún dato que le sirva para identificar el edificio seleccionado. Con este propósito, el mismo objeto `Intent` dispone internamente de campos de tipo diccionario, accesibles mediante los métodos `putExtra(...)`, que permiten asociar a una clave de tipo `String` distintos tipos de información, desde datos primitivos como variables `int` o `boolean`, hasta cualquier objeto que implemente las interfaces `Serializable` o `Parcelable`.

Así, para enviarle a la actividad Plano el identificador del `Edificio` seleccionado, basta con utilizar el método `putExtra(String, int)` de la clase `Intent` para incluir en la llamada el valor, identificado con una clave de tipo `String` definida por la propia clase `GUPlano`:

```
i.putExtra(GUPlano.KEY_EDIFICIO, mIndexEdificio);
```

Hecho esto, en el momento en el que se ejecuta la llamada a `startActivity(Intent)` y el `Intent` llega a la actividad Plano, la clase `GUPlano` recupera durante su inicialización el valor del índice del `Edificio` y lo utiliza para cargar el plano adecuado:

```
if (getIntent().getExtras() != null) {
      mIndexEdificio = getIntent().getExtras().getInt(KEY_EDIFICIO);
}
if (mIndexEdificio != -1) { cambiaImagenPlano(view); }
```

Lo mismo sucede cuando queremos enviar a la actividad Mapa la información necesaria para que centre el mapa en un edificio concreto. En este caso, la clase `GUMapa` espera recibir, por una parte, un `boolean` que le indique si debe centrar el mapa en unas coordenadas en concreto y, por otra, las propias coordenadas. El envío de estos datos se hace mediante los métodos `putExtra(String, boolean)` y `putExtra(String, int)` de la siguiente forma:

```
Intent i = new Intent(getApplicationContext(), GUMapa.class);
i.putExtra(GUMapa.KEY_CENTRAR, true);
i.putExtra(GUMapa.KEY_LAT, edificio.getCoordenadas().getLatitudeE6());
i.putExtra(GUMapa.KEY_LON, edificio.getCoordenadas().getLongitudeE6());
startActivity(i);
```

Y la recepción de los datos se lleva a cabo en la clase `GUMapa` utilizando los métodos `getBoolean(String)` y `getInt(String)` del objeto `Bundle` que guarda los valores del `Intent`:

```
Bundle extras = getIntent().getExtras();
if (extras != null && extras.getBoolean(KEY_CENTRAR) == true) {
      GeoPoint centro = new GeoPoint(extras.getInt(KEY_LAT),
                                     extras.getInt(KEY_LON));
      centrarMapa(centro);
}
```

Como vemos, la primera forma de recibir datos, mediante información extra en el objeto `Intent`, es un mecanismo bastante sencillo y a la vez efectivo.

Ciclo de vida de una actividad

La otra forma es la utilizada por la propia actividad para enviarse información a sí misma. ¿A sí misma? Así es: a lo largo del ciclo de vida de una aplicación Android, es muy normal que las actividades pasen varias veces de un estado activo, en ejecución, a un estado inactivo, que puede ser una pausa momentánea o un cese de ejecución más

prolongado. Concretamente, los estados pueden ser principalmente tres:

- **En ejecución**: la actividad está en primer plano y funcionando normalmente.

- **En pausa**: aunque hay otra actividad ejecutándose en primer plano, esta todavía es visible. Es decir, la actividad que está en primer plano tiene una zona transparente o bien no ocupa toda la pantalla, permitiendo que al menos una parte de la otra actividad siga siendo visible. Las actividades que están en pausa se mantienen en memoria, conservan su estado y son tenidas en cuenta por el gestor de ventanas, pero el sistema operativo puede finalizar su ejecución en situaciones de escasez extrema de memoria.

- **Detenida**: la actividad no es visible, está completamente oculta por otra actividad distinta y ha pasado a segundo plano. El objeto `Activity` de esta actividad permanece en memoria, pero no es tenido en cuenta por el gestor de ventanas y el sistema operativo puede finalizar su ejecución en cualquier momento si otras aplicaciones necesitan la memoria que está ocupando.

Supongamos que estamos usando una aplicación y entra una llamada de teléfono. La interfaz para aceptar o rechazar la llamada es una actividad como otra cualquiera, y en ese momento es la que tiene el foco principal; mientras dure la conversación, la primera aplicación, que ha quedado oculta por la llamada de teléfono, se detiene y pasa a segundo plano, a la espera de ser retomada. Esta actividad estaría detenida. En otras ocasiones, puede que nos aparezca en pantalla una notificación (como cuando llega un mensaje de texto nuevo) que, aunque ocupa el primer plano, no llega a ocultar a la aplicación que estábamos usando. En este caso, la primera aplicación quedaría en un estado de pausa.

Estos cambios de estado los gestiona el sistema operativo automáticamente según las acciones del usuario y las necesidades de memoria y rendimiento que tenga el sistema en ese momento, pudiendo el sistema finalizar la ejecución de aplicaciones que estén detenidas en segundo plano si otras aplicaciones o servicios necesitan recursos adicionales.

Cuando esto sucede, a las actividades se les da la oportunidad de almacenar su estado actual; esto es, cualesquiera datos que podrían perderse al pasar la aplicación a un estado de suspensión y que sean necesarios para continuar la tarea actual en el punto en el que se encuentra. Al producirse un cambio de estado, el sistema operativo efectúa llamadas a los métodos `onPause()`, `onStop()`... que son heredados por cualquier subclase de `Activity` y que pueden utilizarse para efectuar el guardado del estado de la actividad antes de que el sistema la pase a segundo plano; además, también se ofrece el método `onSaveInstanceState(Bundle)` que el sistema asegura que será llamado siempre que se vaya a producir una transición de estado en la cual es posible que la actividad necesite guardar estos datos.

Por ejemplo, una aplicación que permite redactar notas, al detectar que pasa a un estado de pausa o detención podría guardar el texto introducido hasta el momento, de manera que si entra una llamada de teléfono el usuario no tenga que volver a escribir el texto desde el principio una vez que la actividad vuelva a primer plano. Otro ejemplo, ya de nuestra aplicación, sería el identificador del `Edificio` sobre el que se esté trabajando en ese momento: ante un cambio de estado, las actividades deben guardar dicha información para asegurar que, cuando vuelvan a primer plano, el edificio mostrado seguirá siendo el mismo.

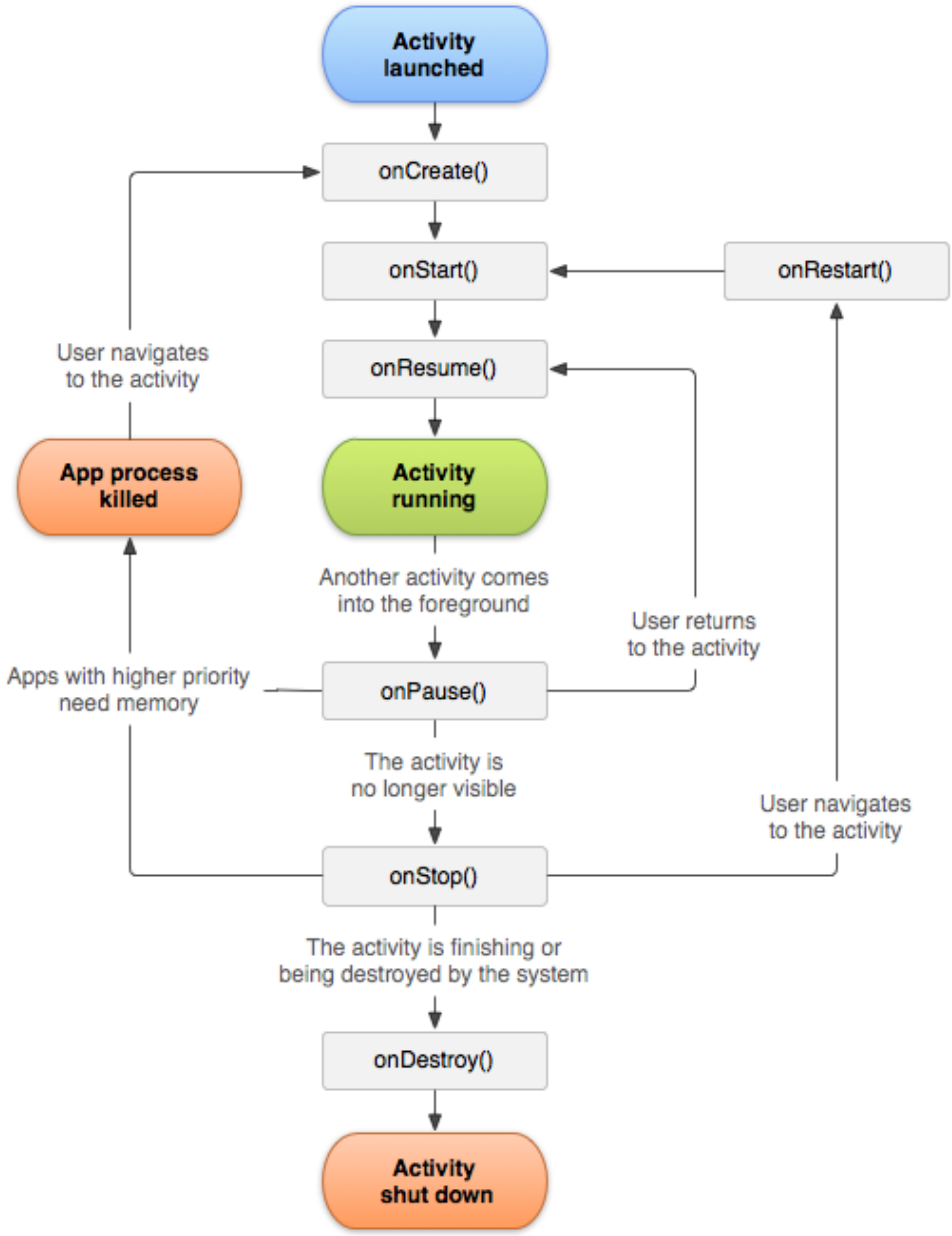

Figura 1. Ciclo de vida de una actividad Android, indicando las llamadas que hace el sistema operativo en cada momento para notificarnos los cambios de estado.

Control del impacto de los cambios de estado en la funcionalidad

En lo que toca a nuestra aplicación, una de las actividades que más notablemente se ven afectadas por estos cambios de estado, es la actividad Mapa. Los dispositivos Android tienen la característica de que permiten su utilización con diferentes orientaciones de pantalla, ya sea con el dispositivo en orientación vertical u horizontal; y aunque el sistema operativo gestiona automáticamente estos cambios a nivel gráfico, durante las pruebas posteriores al desarrollo decidimos verificar que todas las actividades funcionaban adecuadamente tanto en una orientación como en otra.
En el caso de la actividad Mapa, durante las pruebas se observó que, si cambiábamos la orientación de la pantalla con la actividad en su estado inicial, no había problema; pero si efectuábamos la misma operación teniendo una ruta dibujada en pantalla, la ruta desaparecía y la actividad volvía a su estado inicial. Tras investigar el error se descubrió que estaba causado por una mala gestión de los cambios de estado: por lo visto, al girar el dispositivo, el sistema operativo tiene que detener la actividad para reconfigurar la pantalla, volviendo a ejecutar la actividad con la nueva orientación. Como nuestra actividad no estaba guardando en ningún momento la información de la ruta mostrada, al relanzarse la actividad mediante llamada al método onCreate(Bundle) volvía a ejecutarse toda la lógica desde el principio sin tener en cuenta las últimas acciones, volviendo la actividad a su estado inicial.

Para solucionar este error, se sobrescribió el método `onSaveInstanceState(Bundle)` de manera que al ser ejecutado por el sistema operativo, la clase `GUMapa` almacenase la información de su estado: un indicador de si había alguna ruta mostrándose en pantalla, y los datos necesarios para reconstruir dicha ruta (origen y destino).

```
@Override
protected void onSaveInstanceState(Bundle outState) {
      outState.putBoolean(GUMapa.KEY_RUTAVISIBLE, mRutaVisible);
      outState.putString(GUMapa.KEY_RUTADESTINO, mRutaDestino);
      outState.putString(GUMapa.KEY_RUTAORIGEN, mRutaOrigen);
      super.onSaveInstanceState(outState);
}
```

El objeto `outState` de tipo `Bundle` provisto por el sistema para guardar los datos necesarios, es el mismo que se incluirá posteriormente como parámetro del método `onCreate(Bundle)` a la hora de relanzar la actividad de nuevo.

Así, la clase `GUMapa` puede comprobar al inicio si existe alguna información de su estado anterior y retomarlo; en este caso lo que hacemos es verificar si existe dicho objeto Bundle y, de ser así, si en el mismo se indica que hay que mostrar una ruta.

```
public void onCreate(Bundle savedInstanceState) {
      // ...
      if (savedInstanceState != null) {
            mRutaVisible = savedInstanceState
                        .getBoolean(GUMapa.KEY_RUTAVISIBLE);
            if (mRutaVisible) {
                  mRutaDestino = savedInstanceState
                              .getString(GUMapa.KEY_RUTADESTINO);
                  mRutaOrigen = savedInstanceState
                              .getString(GUMapa.KEY_RUTAORIGEN);
                  mEdificiosOvr.pintarRuta(mRutaDestino, mRutaOrigen);
            }
      }
```

Si la respuesta es afirmativa entonces recuperamos los datos de origen y destino de la ruta y volvemos a calcularla y pintarla para que el usuario no se vea afectado por el cambio de orientación.

De esta manera, controlando las notificaciones de cambio de estado que nos hace el sistema operativo, podemos conseguir que nuestras aplicaciones mantengan su estado a lo largo de todo su ciclo de vida sin que la experiencia del usuario se vea afectada.

Fortalezas, carencias y mejoras

A lo largo de esta memoria hemos explicado el proceso de desarrollo de la aplicación GUÍA UPCT para dispositivos móviles bajo el sistema operativo Android. En este capítulo vamos a proceder a analizar cuáles son los puntos fuertes del producto final, qué características del programa son aquellas que consideramos que le dan valor de cara al usuario y qué decisiones de diseño le proporcionan ventaja sobre otras posibles alternativas. De igual forma, revisaremos también aquellos aspectos de la aplicación que se puedan considerar mejorables, ya sea a nivel teórico, técnico o de usabilidad, y posibles maneras de perfeccionarlos.

Punto fuerte: Compatibilidad multidispositivo

Un aspecto remarcable de nuestra aplicación es que, por el simple motivo de haber elegido Android como plataforma objetivo, la aplicación GUÍA UPCT puede ejecutarse sin cambios en cualquier dispositivo que utilice el sistema operativo Android, lo cual nos abre un gran abanico de posibilidades.

Cuando se comenzó el desarrollo de la aplicación, la presencia de Android en el mercado de dispositivos móviles estaba limitada a unos cuantos modelos de *smartphone*; en el escenario actual, en cambio, además de multiplicarse el número de modelos de teléfono, han irrumpido con fuerza también los dispositivos de tipo *tablet*, con una buena aceptación por parte de los usuarios y multitud de fabricantes ofertando modelos de tabletas basadas en Android, y la rápida expansión de éste hace que podamos obtener incluso ordenadores portátiles que lo usan como único sistema operativo. Pues bien, gracias a haber elegido Android como plataforma base, decíamos, nuestra aplicación podría funcionar sin problemas en cualquiera de todos esos dispositivos, facilitando el llegar a un mayor número de usuarios de diferentes perfiles.

Si en vez de desarrollar una aplicación para Android la hubiéramos desarrollado para iOS, por ejemplo, nuestro público objetivo quedaría reducido solamente a aquellos usuarios que posean un dispositivo iPod, iPhone o iPad; dado que estos modelos, fabricados exclusivamente por Apple, suelen situarse en el rango superior de precios dentro de sus respectivos segmentos, se puede esperar que no sean tan frecuentes entre alumnos de primeros años y al elegirlo estaríamos limitando artificialmente el número de posibles usuarios.

Punto fuerte: Extensibilidad a base de actividades

Otra de las ventajas de desarrollar la GUÍA UPCT sobre Android, es que resulta muy sencillo añadirle nuevas funcionalidades y pantallas: basta con implementar una nueva Actividad que lleve a cabo la funcionalidad que queremos, y llamarla desde cualquier otra de las actividades existentes en la aplicación, mediante el `Intent` correspondiente.

Por ejemplo, si quisiéramos añadir la posibilidad de que el usuario pueda cambiar la

descripción de un edificio, podríamos crear una Actividad que reciba al iniciarse un texto, presente al usuario una pantalla para modificarlo, y al terminar envíe de vuelta dicho texto; incluso podríamos usar una ya existente. Desde la Ficha únicamente tendríamos que crear un `Intent` para esta nueva pantalla, pasarle el texto de la descripción e indicarle que queremos recibir el texto modificado de vuelta. Para permitir que el usuario personalice la fotografía de los edificios, podríamos llamar a la actividad Cámara ya existente, etc. Haciéndolo así, conseguimos que desde el punto de vista del usuario la aplicación tenga nuevas pantallas integradas, aunque programáticamente no hayamos tenido que implementarlas nosotros.

En otras plataformas, este mecanismo no es tan sencillo; con arquitecturas pensadas para aplicaciones completas que se lanzan desde un menú, donde no está previsto que una aplicación pueda utilizar "partes" de otra aplicación cualquiera, el concepto de actividad reutilizable entre aplicaciones no tiene un equivalente claro y se hace necesario diseñar algunos de los componentes en forma de librería si se quiere obtener la misma versatilidad.

Punto fuerte: Posibilidad de uso offline

En la sección dedicada a la actividad Ficha, comentábamos cómo la dependencia de la conexión de datos nos había llevado a idear funcionalidad alternativa a la que se pudiera acceder indistintamente con conexión o sin ella. Las actividades y flujo de navegación resultantes de este cambio de enfoque, consideramos que son también uno de los puntos fuertes de la GUÍA UPCT.

La conexión permanente del dispositivo a Internet es una opción que, incluso aunque eliminemos el coste económico conectando a redes WiFi disponibles (como la red WiFi UPCT), sigue teniendo un coste para el usuario en términos de duración de la batería; puede darse, por tanto, el caso de usuarios que aún teniendo la posibilidad de activar la conexión de datos, prefieren no hacerlo para ahorrar energía durante el uso habitual del terminal.

Para este tipo de usuarios, nuestra aplicación sigue ofreciendo funcionalidad útil: se sigue permitiendo el acceso a los planos de los edificios y a la descripción básica de los POI de cada uno de ellos, con lo cual el usuario puede utilizar la GUÍA UPCT para orientarse dentro de un edificio del campus sin tener que conectarse a Internet; y se sigue teniendo también acceso --a través de la Ficha-- a los datos almacenados sobre cada uno de los edificios, sirviendo como referencia informativa sobre el entorno del campus de la Universidad Politécnica de Cartagena.

A mejorar: Dificultad de implementar un patrón MVC puro

Antes de comenzar con el desarrollo de la aplicación, nos fijamos el objetivo de utilizar en el diseño de la misma el patrón Modelo-Vista-Controlador, a fin de poder tomarlo como ejemplo didáctico de implementación de un patrón de diseño conocido en un programa real.

Empero, al tratar de aplicar el patrón de manera efectiva durante el desarrollo, hemos visto que el diseño de la arquitectura de Android no facilita esta tarea, debido sobre todo al alto acoplamiento existente entre el Controlador y la Vista en algunas de las clases básicas, como por ejemplo las derivadas de `View`. Al no poder implementar correctamente el patrón MVC, el objetivo de utilizar nuestro código como material didáctico representativo no se ve satisfecho en su totalidad.

Mejora: Uso de patrones derivados

Una forma de paliar esta carencia del proyecto sería cambiando el punto de vista: se puede aprovechar nuestra implementación para dar a conocer al alumno patrones derivados de MVC, como son el patrón Modelo-Vista-Presentador (MVP) y el patrón Modelo-Vista-VistaModelo (MVVM), ya mencionados en el capítulo dedicado a los patrones de diseño y que representan una buena aproximación del patrón MVC en condiciones en las que la aplicación de un patrón MVC "puro" es complicada.

A mejorar: Datos estáticos

De todas las decisiones de diseño tomadas durante el desarrollo de la aplicación, quizá la menos elegante de todas sea la de almacenar la información de los edificios de manera estática en el propio código fuente de la clase `DatosAplicacion`. Se optó por implementar este aspecto de esta manera para no alargar ni complicar el desarrollo del proyecto más allá de lo necesario, siendo la opción más rápida en las fases iniciales; pero su desventaja es que dificulta sobremanera la actualización o modificación de los datos de los edificios, viéndonos obligados a editar y recompilar el código fuente para ello o, peor aún, a publicar una nueva versión de la aplicación (un nuevo paquete `.apk`) cada vez que fuera necesario cualquier cambio en los datos de los edificios, desde la descripción hasta la foto, pasando por los puntos de interés.

Mejora: Obtención de datos dinámica vía XML o web

Para contrarrestar esta limitación, lo ideal sería redefinir la lógica de la clase `DatosAplicacion` para que efectúe la carga de los datos de los edificios de manera dinámica, a partir de algún origen externo: por ejemplo, un fichero XML o un servicio web.
Usando un fichero XML, aunque sea desde el mismo directorio de recursos de la aplicación, conseguimos de cara al programador estructurar la información disponible de los edificios de una manera más clara, y evitamos que cambios en la misma provoquen recompilaciones del código. Si por el contrario este fichero XML se encontrara en un servidor remoto y pudiera descargarse en caso necesario, podríamos además tener información completa y actualizada sin necesidad de distribuir nuevas versiones de la aplicación.

A mejorar: Falta de opciones de personalización

Otra posible carencia, la última que vamos a analizar, es la falta de opciones para configurar la aplicación a gusto personal del usuario. Para una prueba de concepto, estas funcionalidades podrían considerarse superfluas, pero producto final lo suficientemente refinado sí que debería admitir un cierto grado de personalización y de opciones, so pena de ofrecer un aspecto de aplicación fría y poco interactiva, con una línea de uso demasiado plana.

Mejora: Opciones configurables

Existen varias funcionalidades de la GUÍA UPCT que podrían ofrecer al usuario cierto control sobre el comportamiento predeterminado, algunas de estas opciones serían:

- Mapa
  - A nivel gráfico: elección del color de la ruta dibujada en pantalla; elección del icono de los edificios; elección de mapa satélite o callejero...
  - A nivel de comportamiento: cálculo de rutas a pie o en coche; cambio de la acción efectuada al pulsar sobre el icono de un edificio...
- Listado
  - A nivel gráfico: mostrar o no mostrar las imágenes en el listado...
  - A nivel de comportamiento: cambio de la acción efectuada al pulsar un elemento de la lista (abrir Ficha o abrir Mapa)...

Para controlar todas estas opciones se podría añadir una nueva Actividad, que muestre al usuario las opciones disponibles y le permita cambiarlas, adaptando la aplicación a su gusto personal. Estos datos de configuración podrían almacenarse mediante el uso de SQLite, el motor de base de datos integrado en Android y el estándar usado por multitud de aplicaciones para guardar sus datos.

Posibles ampliaciones

Consideramos que el proyecto de la aplicación GUÍA UPCT, por su utilidad y por la relevancia en el mundo actual de las tecnologías que utiliza, es susceptible de ser usado como base para elaborar nuevos proyectos de fin de carrera por parte de aquellos alumnos y profesores interesados en el desarrollo de aplicaciones móviles y el mundo Android. A continuación se presentan algunas posibles vías de ampliación que se pueden seguir con este fin.

Posicionamiento dentro de un edificio por triangulación de puntos de acceso

Esta ampliación consistiría en dotar a la aplicación de la capacidad de mostrar al usuario su posición, más o menos exacta, dentro de cualquiera de los edificios del campus.

En su concepción actual, la aplicación GUÍA UPCT se limita a mostrar la ubicación del usuario sólo en la actividad Mapa; es decir, sólo cuando el usuario se encuentra en el exterior de los edificios. Cuando el usuario accede con su dispositivo Android a alguno de los edificios del campus, la API de Google Maps no puede mostrarle el interior del edificio, con lo cual lo normal es que el usuario lance la actividad Plano para consultar el plano interior del edificio y la localización de los puntos de interés del mismo. En este escenario, es el usuario el que debe ser capaz de interpretar el plano para determinar su ubicación dentro del edificio y proximidad a dichos puntos.

Con la ampliación propuesta, la aplicación podría mostrar al usuario su posición también dentro del edificio, no sólo en el exterior. Para conseguirlo, las modificaciones principales a realizar serían las siguientes:

- **Nueva lógica de posicionamiento por triangulación WiFi**. Como hemos comentado anteriormente, la Universidad dispone de una red WiFi que da cobertura a prácticamente todo el campus, existiendo en cada edificio varios puntos de acceso WiFi para ello. Mediante las funcionalidades de conectividad WiFi del dispositivo Android, y controlando en cada momento tanto los puntos de acceso dentro del alcance del dispositivo como aquellos puntos de acceso que hayan sido "vistos" recientemente, se puede triangular la posición del dispositivo dentro del propio edificio y señalizar esta ubicación en la actividad Plano.
- **Añadir detección de proximidad/acceso a edificios**. Esta modificación serviría para que la aplicación pueda diferenciar entre el modo de posicionamiento "externo" basado en los servicios habituales de Android, y el modo de posicionamiento "interno" específico para los edificios del campus, explicado en el punto anterior.
- **Señalización dinámica de la posición del usuario en la actividad Plano**. Esta modificación consistiría simplemente en dotar a la actividad Plano de un nuevo marcador, que se mostraría en las coordenadas del plano

correspondientes a la ubicación del dispositivo en el edificio, y cuya posición se iría actualizando dinámicamente conforme el usuario fuera variando su posición (el intervalo de actualización sería configurable para evitar un gasto de batería innecesario).

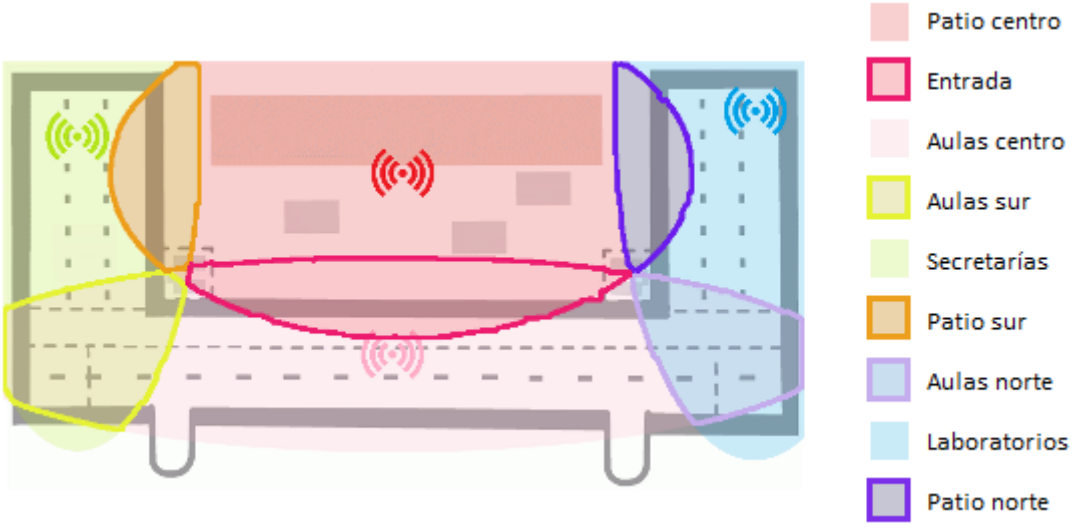

Figura 1. Zonas en las que se podría ubicar a un usuario dentro del edificio de Antigones, suponiendo cuatro puntos de acceso en las posiciones indicadas.

Consideramos que esta es una ampliación muy interesante, tanto desde el punto de vista académico por implicar algoritmos de triangulación de redes, como desde el punto de vista del usuario, al que le será útil para localizar con mayor facilidad los puntos de interés de un edificio que esté visitando por primera vez.

***ACTUALIZACIÓN**: A finales de noviembre del 2011, con la idea anterior ya plasmada en el borrador de esta memoria, Google anunció que iba a incluir en Google Maps (en su versión para Android) la capacidad para mostrar mapas del interior de los edificios y la posición del usuario dentro de los mismos. Aunque esta funcionalidad es la misma que hemos comentado y podría parecer que convierte la ampliación en superflua, no es así: los mapas de los edificios sólo se incluirán en Google Maps mediante petición expresa, y por tanto la ampliación descrita seguirá siendo útil mientras la Universidad Politécnica de Cartagena no solicite a Google que cartografíe el interior de sus edificios. Sirva este dato como muestra de lo imparable del progreso de las aplicaciones móviles en general y de Android en particular.*

Alarmas de tareas pendientes al detectar la cercanía de un edificio

La ampliación propuesta en este apartado, añadiría a la aplicación la funcionalidad de lista de tareas, con la capacidad de asociar cada tarea a un edificio concreto y mostrar

un aviso si el usuario se encuentra cerca de un edificio con tareas pendientes.

Conforme avanza por su camino universitario, el usuario no sólo va conociendo los edificios del campus y los servicios más habituales de cada uno, sino que es probable que de vez en cuando tenga que moverse entre varios edificios para llevar a cabo todas sus gestiones y actividades universitarias (clases, tutorías, exámenes, cursos de idiomas, actividades deportivas, solicitud de becas, actos académicos, presentaciones...). En esta etapa, la funcionalidad actual de la aplicación GUÍA UPCT puede quedarse algo corta para el usuario veterano.

Esta carencia puede suplirse con la ampliación que aquí presentamos, mediante la cual el usuario puede aprovechar las capacidades de geoposicionamiento de la GUÍA UPCT para recordar posibles tareas que tenga que llevar a cabo en el edificio en el que se encuentre.
Por ejemplo, un usuario podría crear una tarea asociada al edificio del Rectorado para comprobar si se ha abierto una determinada convocatoria de becas; si otro día cualquiera el usuario se acerca al Rectorado para hablar con un profesor que tenga allí su despacho, la aplicación (activa en segundo plano) le avisaría de que tiene que revisar la convocatoria, ahorrando al usuario un viaje extra.

Para poder realizar esta ampliación, las modificaciones que se deben acometer son múltiples, tanto a nivel visual como de estructura y funcionalidad de la aplicación, pues al contrario que en la amplicación anterior, en ésta se introducen elementos nuevos (base de datos) que actualmente no se están utilizando.

Algunas de las modificaciones a llevar a cabo serían las siguientes:

- **Añadir actividad para crear y gestionar avisos**. La aplicación tendrá que disponer de nuevas pantallas que permitan crear nuevas alarmas, así como consultar, modificar y eliminar las ya existentes. Todos estos datos serán accesibles mediante SQLite. También será necesario añadir los menús o botones que den acceso a dicha funcionalidad desde las pantallas adecuadas, por ejemplo desde las actividades Ficha o Plano.
- **Añadir funcionalidad en segundo plano (servicio)**. Para llevar a cabo esta ampliación sin que la usabilidad se vea afectada, la aplicación tendrá que ser capaz de ejecutarse en segundo plano, como un servicio, de manera que el usuario pueda utilizar su dispositivo libremente mientras no sea necesario mostrar ningún aviso de tarea. Este servicio será el encargado de mostrar las alarmas al usuario, por lo que tendrán que investigarse los diferentes medios de notificación disponibles en Android (diálogo emergente, sonido, vibración) e implementar la lógica necesaria para utilizarlos convenientemente.
- **Añadir detección de proximidad de edificios**. Igual que en la ampliación anterior, para esta funcionalidad la GUÍA UPCT tiene que ser capaz de detectar que el usuario se encuentra cerca (o dentro) de un edificio del campus, para consultar la base de datos de tareas pendientes y mostrar los avisos oportunos.

En el plano técnico, esta ampliación tiene un interés múltiple, pues conlleva: utilización de bases de datos, creación de nuevas interfaces gráficas y actividades, creación de un servicio en segundo plano, acceso a funcionalidades del dispositivo no utilizadas hasta ahora (notificaciones)... Además conseguiría aumentar la vida útil de la aplicación, dándole valor añadido para usuarios que ya conocen el campus lo suficiente como para necesitar indicaciones a la hora de ir de un edificio a otro.

Conclusiones

Llegamos al final de esta memoria y de este proyecto. Hemos desarrollado una aplicación nueva, para unos dispositivos y un mercado que están en pleno auge y constante evolución, eligiendo una plataforma, Android, de reciente creación pero tremendo éxito. Hemos explicado el escenario actual, las técnicas y herramientas utilizadas y todo el proceso de construcción, y hemos evaluado los puntos fuertes y los no tan fuertes del resultado final. Es el momento de sacar conclusiones; de ver qué hemos aprendido con este proyecto, que podamos aplicar en un futuro a nuestro trabajo.

Facilidad de desarrollo en Android

Desarrollar aplicaciones para Android es fácil. Sin lugar a dudas, esa sería la primera conclusión que nos viene a la cabeza tras evaluar todo el proceso de formación previa, obtención de las herramientas necesarias y programación en sí misma.

En lo que a formación respecta, el equipo de Android en Google ha hecho un excelente trabajo de documentación de su plataforma, con artículos y ejemplos que abarcan todos los puntos de vista: desde el general del usuario que sólo quiere conocer las posibilidades de su dispositivo, hasta los detalles técnicos sobre gestión de memoria que necesita un programador para optimizar el rendimiento de su aplicación. Además, la utilización de tecnologías y estándares abiertos ha propiciado el surgimiento de una cantidad impresionante de sitios web, blogs, foros e incluso canales de YouTube y *podcasts* dedicados a la programación en Android; aficionados, desarrolladores independientes y empresas por igual comparten técnicas y conocimiento que facilitan enormemente los primeros pasos y el aprendizaje de cualquiera que esté interesado.

Las herramientas *software* necesarias, ya hemos visto anteriormente que están disponibles a un coste cero: kit de desarrollo y librerías Android, librerías Java, editor Eclipse integrado con el *plug-in* ADT... Todas ellas herramientas profesionales con características muy avanzadas, compatibles con casi cualquier ordenador personal (cosa que no sucede con otras plataformas) y que podemos obtener con unos simples clics de manera gratuita. Pero ¿y las herramientas *hardware*? Desarrollar aplicaciones usando simuladores está muy bien, pero siempre es recomendable disponer físicamente de un terminal en el que poder probar los programas. Afortunadamente, la variedad de fabricantes y operadores que han dado su apoyo al sistema Android hace que hoy día sea realmente fácil conseguir un dispositivo, ya sea un teléfono móvil o un dispositivo de tipo *tablet*, por un precio muy reducido en comparación con los precios de terminales de otros sistemas, como Blackberry o iPhone.

La facilidad del proceso de codificación en sí mismo es una característica algo más subjetiva, puesto que depende de los conocimientos y la destreza de cada programador. Sin embargo, aquí Java parte con ventaja: no solo es uno de los lenguajes más utilizados tanto en entornos académicos como productivos, sino que de hecho, cuando apareció Android, Java llevaba ya casi 10 años siendo utilizado por los

desarrolladores de aplicaciones móviles para Symbian; con lo que es muy probable que cualquier programador que se inicie en Android tenga ya experiencia previa en Java. Otros factores, como la modularidad que proporcionan las Actividades o la sencillez de creación de las interfaces gráficas mediante XML, contribuyen también a esta facilidad de codificación.

Todo lo anterior unido da como resultado que un programador medio, sin experiencia previa en desarrollo de aplicaciones móviles, pueda montar todo el entorno necesario, diseñar y codificar su primera aplicación Android, con funcionalidad real, en un fin de semana. Y algo que no hemos comentado hasta ahora: todo este proceso puede llevarse a cabo libremente. Google no requiere que los desarrolladores de aplicaciones Android se registren como tales, salvo que quieran publicar sus aplicaciones en Android Market; así, cualquiera puede obtener acceso a todas las herramientas y documentación necesarias, hacer sus primeras aplicaciones de prueba o incluso diseñar una aplicación para un proyecto :-) sin necesidad de darle sus datos a ninguna empresa ni pagar ningún tipo de cuota.

Viabilidad de una aplicación mashup

La segunda conclusión que obtenemos, y que tiene que ver con los objetivos planteados inicialmente, es que el concepto de aplicación *mashup* es perfectamente viable. En el caso que nos ocupa, a modo de concepto, hemos tomado dos servicios o aplicaciones ya existentes: Google Maps y un visor de imágenes, y los hemos combinado con los datos de los edificios de la UPCT para construir una aplicación nueva que ofrece a los alumnos y usuarios del campus funcionalidad realmente útil. Es el concepto de reutilización llevado al extremo: en vez de reutilizar sólo los algoritmos o las librerías, integrar directamente una aplicación con otra para obtener los resultados que se desean.

Dependencia de la conexión de datos

La tercera conclusión es el resultado de la experiencia de usar un dispositivo Android durante los meses que ha durado el desarrollo de este proyecto; una conclusión que, aunque se puede interpretar como una crítica, en realidad no es más que la constatación del escenario en que vive actualmente el mundo de las telecomunicaciones: dependencia de la conexión de datos.

Efectivamente, un dispositivo Android que no tiene una conexión de datos activa, está tremendamente limitado pues la mayoría de aplicaciones dependen de ella para obtener al instante información dinámica y personalizada. Una aplicación puede activar el GPS para indicar la ubicación del dispositivo pero, si no dispone de conexión de datos, no podrá descargar un mapa de la zona ni buscar gasolineras cercanas. Nuestra propia aplicación puede decirnos que el POI seleccionado en la actividad Plano es la cafetería pero, sin conexión de datos, no podremos acceder a la web para

consultar el menú.

En este aspecto, el mercado debe evolucionar para que la conexión de datos en los dispositivos móviles sea una característica por defecto y accesible por todos los usuarios, procurando ofrecer planes de precios ajustados al coste y consumo reales, ya que es la única forma de sacarle todo el partido a un dispositivo de estas características.

Publica rápido

La última conclusión es una lección a futuro.

A lo largo del desarrollo de este proyecto, que por motivos laborales y personales se ha extendido durante más de un año, han aparecido 5 nuevas versiones del sistema operativo Android (desde la 2.3 hasta la 4.0) y se ha dado el salto a las *tablets*, algo que en 2010 era tan sólo un rumor. Si a principios de año en la UPCT no había ningún proyecto de fin de carrera relacionado con Android, hoy tenemos incluso ciclos de charlas dedicados al tema. Funcionalidad que habíamos ideado como posible ampliación de la aplicación GUÍA UPCT ha sido ya incorporada por Google a su API de Google Maps como funcionalidad nativa...

La lección que debemos extraer de todo esto es que, en el mundo de las aplicaciones móviles al menos, hay que ser rápido. Una idea que hoy es novedosa, puede que dentro de tres o seis meses la hayan implementado de diez maneras distintas. Por eso, con tecnologías que evolucionan a tanta velocidad, si queremos que nuestros proyectos marquen una diferencia tenemos que estar preparados para enfocar nuestros esfuerzos a obtener una primera versión funcional lo más rápidamente posible. Ya en 1997, el mismo año que surgió el primer teléfono móvil con aplicaciones incorporadas, Eric S. Raymond enunció el paradigma "*release early, release often*" (publica pronto, publica a menudo) según el cual es preferible publicar una primera versión de nuestras aplicaciones lo antes posible, e ir corrigiendo los errores con posterioridad, sacando nuevas versiones tan a menudo como sea necesario. A lo largo de este proyecto hemos constatado que, ciertamente, en lo que a aplicaciones móviles se refiere, este principio es de mayor relevancia que nunca y debe ser tenido en cuenta a la hora de afrontar cualquier proyecto de similares características.

Resumen

En resumen, a lo largo de este proyecto hemos conseguido crear una aplicación móvil de utilidad para la comunidad universitaria de la UPCT, que gracias a la elección de la plataforma Android podrá ser utilizada en una mayor variedad de dispositivos y a un coste mucho más reducido que el de otras alternativas. Al aprovechar servicios ya

existentes, el desarrollo ha sido sencillo y ofrece un resultado más depurado que si se hubiera programado toda la lógica desde cero; y, aunque también ha sido rápido, hemos observado que en lo que a aplicaciones móviles se refiere es mejor seguir una filosofía de publicación temprana que nos permita adelantarnos a los avances del mercado, ya que es un entorno en continua evolución.

Bibliografía

Utilizada durante la fase de desarrollo

Deitel, Paul J.; Deitel, Harvey M. *Java: How to Program*. 9th ed. Estados Unidos: Prentice Hall, 2012. ISBN 978-0-13-257566-9.

Universidad Politécnica de Cartagena: *La Universidad* [en línea]. Disponible en <http://www.upct.es/contenido/universidad/index_universidad.php>. Consultado en febrero de 2011.

IBM developerWorks: *Introduction to Android development* [en línea]. Disponible en <http://www.ibm.com/developerworks/opensource/library/os-android-devel/>. Consultado en febrero de 2011.

Android Developer: *The Developer's Guide* [en línea]. Disponible en <http://developer.android.com/guide/index.html>. Consultado en febrero de 2011.

StackOverflow: *Newest 'android' questions* [en línea]. Disponible en <http://stackoverflow.com/questions/tagged/android>. Consultado en marzo de 2011.

Google Code: *Google Maps API Family* [en línea]. Disponible en <http://code.google.com/intl/es/apis/maps/>. Consultado en mayo de 2011.

Google Code: *Preguntas frecuentes sobre KML* [en línea]. Disponible en <http://code.google.com/intl/es/apis/kml/faq.html>. Consultado en abril de 2011.

Lars Vogel: *Location API and Google Maps in Android – Tutorial* [en línea]. Version 1.5, 2011. Disponible en <http://www.vogella.de/articles/AndroidLocationAPI/article.html>. Consultado en marzo de 2011.

Krzysztof Grajek: *Android Series: Custom ListView items and adapters* [en línea]. 2009. Disponible en <http://www.softwarepassion.com/android-series-custom-listview-items-and-adapters/>. Consultado en mayo de 2011.

Ed Burnette: *How to use Multi-touch in Android 2* [en línea]. 2010. Disponible en <http://www.zdnet.com/blog/burnette/how-to-use-multi-touch-in-android-2/1747>. Consultado en abril de 2011.

Android People: *Android Loading Welcome Splash / Spash Screen Example* [en línea]. Disponible en <http://www.androidpeople.com/android-loading-welcome-splash-spash-screen-example>. Consultado en junio de 2011.

Nicolas Gramlich: *Parsing XML from the Net - Using the SAXParser* [en línea]. Disponible en <http://www.anddev.org/parsing_xml_from_the_net_-_using_the_saxparser-t353.html>. Consultado en abril de 2011.

Sobre aplicaciones móviles

GSM Arena: *Samsung I9100 Galaxy S II* [en línea]. Disponible en <http://www.gsmarena.com/samsung_i9100_galaxy_s_ii-3621.php>. Consultado en agosto de 2011.

EveryMac: *Powerbook G4* [en línea]. Disponible en <http://www.everymac.com/systems/apple/powerbook_g4/index-powerbook-g4.html>. Consultado en agosto de 2011.

Jukov, M.: *Mobile applications: the history of the issue* [en línea]. Disponible en <http://www.articledashboard.com/Article/Mobile-applications-the-history-of-the-issue/1048768>. Consultado en septiembre de 2011.

Wikipedia: *Mobile operating system* [en línea]. Disponible en <http://en.wikipedia.org/wiki/Mobile_operating_system>. Consultado en septiembre de 2011.

Wikipedia: *History of mobile phones* [en línea]. Disponible en <http://en.wikipedia.org/wiki/History_of_mobile_phones>. Consultado en septiembre de 2011.

Wikipedia: *PDA* [en línea]. Disponible en <http://es.wikipedia.org/wiki/PDA>. Consultado en septiembre de 2011.

Wikipedia: *Telefonía móvil en España: Historia* [en línea]. Disponible en <http://es.wikipedia.org/wiki/Telefon%C3%ADa_m%C3%B3vil_en_Espa%C3%B1a#Historia>. Consultado en octubre de 2011.

Zafra, Juan Manuel: *El Gobierno quiere cerrar MoviLine para dar dos licencias más de móvil* [en línea]. *EL PAÍS,* ed. 19 de marzo de 2001. Disponible en <http://elpais.com/diario/2001/03/19/economia/984956401_850215.html>. Consultado en octubre de 2011.

Wikipedia: *Wireless Application Protocol* [en línea]. Disponible en <http://en.wikipedia.org/wiki/Wireless_Application_Protocol>. Consultado en octubre de 2011.

Ciberaula: *Introducción a J2ME* [en línea]. Disponible en <http://java.ciberaula.com/articulo/introduccion_j2me>. Consultado en octubre de 2011.

Mahmoud, Qusay: *MIDP for Palm OS 1.0: Developing Java Applications for Palm OS Devices* [en línea]. Disponible en <http://developers.sun.com/mobility/midp/articles/palm/>. Consultado en noviembre de 2011.

Wikipedia: *Smartphone* [en línea]. Disponible en <http://en.wikipedia.org/wiki/Smartphone>. Consultado en noviembre de 2011.

Wikipedia: *Symbian OS* [en línea]. Disponible en <http://en.wikipedia.org/wiki/Symbian_OS>. Consultado en noviembre de 2011.

al-Baker, Asri: *Symbian Device: The OS Evolution* [en línea]. Disponible en <http://www.i-symbian.com/wp-content/uploads/2009/11/Symbian_Evolution.pdf>. Consultado en noviembre de 2011.

Ilyas, Mohammad; Ahson, Syed: *Smartphones Research Report* [en línea]. *IEC*, ed. junio 2006, vol. 2. Disponible en <http://www.iec.org/newsletter/june06_2/analyst_1.html>. Consultado en diciembre de 2011.

Wikipedia: *BlackBerry* [en línea]. Disponible en <http://en.wikipedia.org/wiki/BlackBerry>. Consultado en noviembre de 2011.

Pawlowski, Marek: *Growth in open OS will shift zones of influence* [en línea]. Disponible en <http://www.mobileuserexperience.com/?p=365>. Consultado en diciembre de 2011.

Canalys: *64 million smart phones shipped worldwide in 2006* [en línea]. Disponible en <http://www.canalys.com/newsroom/64-million-smart-phones-shipped-worldwide-2006>. Consultado en diciembre de 2011.

Canalys: *Smart mobile device shipments hit 118 million in 2007, up 53% on 2006* [en línea]. Disponible en <http://www.canalys.com/newsroom/smart-mobile-device-shipments-hit-118-million-2007-53-2006>. Consultado en diciembre de 2011.

Belic, Dusan: *EMEA smartphone market growth rises to 11.7%* [en línea]. Disponible en <http://www.intomobile.com/2006/10/28/emea-smartphone-market-growth-rises-to-117/>. Consultado en diciembre de 2011.

Shah, Neil: *Smartphone War Part I: : Smartphone OS the differentiating factor* [en línea]. Disponible en <http://shahneil.com/2010/02/smartphone-war-part-i-smartphone-os-differentiating-factor/>. Consultado en diciembre de 2011.

IT Facts: *Mobile device OS shares: Symbian – 55.9%, Windows – 12.7%, Linux – 11.3%* [en línea]. Disponible en <http://www.itfacts.biz/mobile-device-os-shares-symbian-559-windows-127-linux-113/4090>. Consultado en diciembre de 2011.

IT Facts: *Smartphone OS shares: Symbian – 76.2%, Linux – 13.7%, PalmOS – 4.6%, Windows – 4.5%, RIM – 1.0%* [en línea]. Disponible en <http://www.itfacts.biz/smartphone-os-shares-symbian-762-linux-137-palmos-46-windows-45-rim-10/4049>. Consultado en diciembre de 2011.

IT Facts: *Smart device OS market shares: Symbian – 62.8%, Microsoft – 15.9%, PalmSource – 9.5%* [en línea]. Disponible en <http://www.itfacts.biz/smart-device-os-market-shares-symbian-628-microsoft-159-palmsource-95/4199>. Consultado en diciembre de 2011.

IT Facts: *Symbian phone shipments up 191% in the first half of 2005* [en línea]. Disponible en <http://www.itfacts.biz/symbian-phone-shipments-up-191-in-the-first-half-of-2005/4500>. Consultado en diciembre de 2011.

IT Facts: *Mobile OS market shares in 2005: Symbian – 51%, Linux – 23%, Windows – 17%* [en línea]. Disponible en <http://www.itfacts.biz/mobile-os-market-shares-in-2005-symbian-51-linux-23-windows-17/5773>. Consultado en diciembre de 2011.

Sobre Android

Wikipedia: *Android (operating system)* [en línea]. Disponible en <http://en.wikipedia.org/wiki/Android_%28operating_system%29>. Consultado en febrero de 2012.

Canalys: *Android takes almost 50% share of worldwide smart phone market* [en línea]. Disponible en <http://www.canalys.com/newsroom/android-takes-almost-50-share-worldwide-smart-phone-market>. Consultado en febrero de 2012.

Green, David: *Android vs. Iphone Development: A Comparison* [en línea]. Disponible en <http://java.dzone.com/articles/android-vs-iphone-development>. Consultado en febrero de 2012.

Android Developer: *Application Fundamentals* [en línea]. Disponible en <http://developer.android.com/guide/topics/fundamentals.html>. Consultado en febrero de 2012.

Sobre patrones de diseño

Metsker, Stephen J.; Wake, William C. *Design Patterns in Java.* 2nd ed. Westford: Addison-Wesley, 2008. ISBN 0-321-33302-0.

Stack Overflow: *Singleton vs. Application Context in Android?* [en línea]. Disponible en <http://stackoverflow.com/questions/3826905/singletons-vs-application-context-in-android>. Consultado en julio de 2011.

Reenskaug, Trygve: *MVC* [en línea]. Disponible en <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>. Consultado en enero de 2012.

Stack Overflow: *MVC pattern in Android?* [en línea]. Disponible en <http://stackoverflow.com/questions/2925054/mvc-pattern-in-android>. Consultado en enero de 2012.

Peckham, James: *MVP in Android* [en línea]. Disponible en <http://jamespeckham.com/Blog/10-11-21/MVP_on_Android.aspx>. Consultado en enero de 2012.

Wikipedia: *Model-view-presenter* [en línea]. Disponible en <http://en.wikipedia.org/wiki/Model–view–presenter>. Consultado en enero de 2012.

Potel, Mike: *MVP: Model-View-Presenter. The Talingent Programming Model for C++ and Java* [en línea]. Disponible en <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>.

Wikipedia: *Model View ViewModel* [en línea]. Disponible en <http://en.wikipedia.org/wiki/Model_View_ViewModel>. Consultado en enero de 2012.

Fowler, Martin: *Supervising Controller* [en línea]. Disponible en <http://martinfowler.com/eaaDev/SupervisingPresenter.html>. Consultado en enero de 2012.

Fowler, Martin: *Passive View* [en línea]. Disponible en <http://martinfowler.com/eaaDev/PassiveScreen.html>. Consultado en enero de 2012.

Otros

Raymond, Eric S. *The Cathedral & the Bazaar*. O'Reilly, 1999. ISBN 1-56592-724-9. Disponible en <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/>. Consultado en marzo de 2012.

Universidad Carlos III de Madrid: *Cómo citar bibliografía* [en línea]. Disponible en <http://www.uc3m.es/portal/page/portal/biblioteca/aprende_usar/como_citar_bibliografia>. Consultado en septiembre de 2011.

Cartagena, a 18 de marzo de 2012.