**UNIVERSIDAD CARLOS III DE MADRID**

**ESCUELA POLITÉCNICA SUPERIOR**

**INGENIERÍA EN INFORMÁTICA**

**PROYECTO FIN DE CARRERA**

# DESARROLLO DE APLICACIONES PARA DISPOSITIVOS MÓVILES SOBRE LA PLATAFORMA ANDROID DE GOOGLE

Autor: Jaime Aranaz Tudela
Tutora: Celeste Campo Vázquez
Enero de 2009

*A Elena, por haberme acompañado*
*todos estos años, en los buenos*
*y malos momentos.*

# Agradecimientos

La finalización del proyecto final de carrera representa uno de esos momentos de la vida en los que merece la pena echar la vista atrás y hacer balance de todo lo pasado. No sólo significa el fruto de varios meses de desarrollo, sino que su entrega simboliza también la conclusión de toda una carrera a través de muchos años de entrega, sacrificio y duro trabajo.

Una vez he llegado hasta aquí, puedo decir que estoy muy orgulloso de este logro y que, a pesar de la denigrante legislación que a mí y a mis ahora colegas nos contempla, me siento miembro de pleno derecho en esta profesión tan especial y relevante como es la ingeniería, en este caso la Ingeniería en Informática. La firme y exigente formación que me ha dado la Universidad Carlos III de Madrid no me permite albergar ninguna duda en esto.

Por otro lado, la obtención de un título universitario no es posible de entender si no se enmarca dentro de un proyecto mayor como lo es tu propia vida, y todos aquellos que te han ayudado en él. Por ello, he de estar agradecido en primer lugar a mis padres, Enrique y María Rosa, por haberme dado unos estudios y haber sabido inculcarme los valores que ahora profeso. También a mi abuela, María Rosa, y a mis hermanos Enrique, Carolina, Javier y Vanessa, cuya influencia me ha permitido llegar a ser la persona que soy.

También gracias a mi tutora Celeste por su orientación y sus consejos durante todo el proyecto. Su disponibilidad y el buen trato tenido siempre conmigo han sido de gran valor para mí.

Gracias igualmente a todos aquellos a los que me he cruzado en la carrera y puedo llamar amigos: David, Fernando, Gonzalo, Jaime, José Luis, Karen, Pedro, Roberto...así como a otros muchos compañeros con los que he tenido el placer de coincidir.

Y, por supuesto, gracias a mi novia Elena por haber estado a mi lado durante todos estos años. Espero haber sabido corresponderte.

# Resumen

Los dispositivos móviles constituyen cada vez más una realidad que ofrece al usuario, en un mismo y reducido aparato, funciones de comunicación y procesamiento de datos que van mucho más allá de las simples llamadas telefónicas o la ejecución de aplicaciones básicas. El gigante de Internet Google ha presentado un nuevo sistema operativo para este tipo de dispositivos, Android, que busca ser una firme alternativa a otros sistemas ya ampliamente extendidos como Symbian o Windows Mobile.

El presente proyecto busca conocer y comprender las características y el funcionamiento de este nuevo sistema operativo, averiguando sus posibilidades y ventajas frente a otras alternativas. Además, abarca el desarrollo completo de una aplicación de nombre *ContactMap*, que permite localizar a los contactos mediante mapas y con la que se busca ilustrar de forma práctica la construcción y naturaleza de las aplicaciones para Android.

# Abstract

Mobile devices are increasingly becoming a reality that offer the user, on the same small appliance, communication functions and data processing that are further beyond phone calls or running simple applications. The Internet giant Google has introduced a new operating system for such devices, called Android, which seeks to be a strong alternative to other systems already widely used as Symbian or Windows Mobile.

This project seeks to know and understand the characteristics and performance of this new operating system, finding out their potential and advantages over other alternatives. Moreover, it covers the complete development of an application called *ContactMap*, which allows the user to locate its contacts using maps and seeks to illustrate construction of applications for Android.

# Índice general

# Índice de figuras

# Índice de Tablas

# Índice de Códigos Fuente

# 1 INTRODUCCIÓN

En las siguientes líneas se hace una breve introducción al presente proyecto, exponiendo cuál es su motivación, qué objetivos son los que persigue y cuáles son los contenidos ofrecidos en esta memoria.

## *1.1 Motivación del proyecto*

Al echar la vista atrás y observar el desarrollo tecnológico que ha experimentado la Humanidad desde mediados del siglo XX hasta hoy, no cabe duda de que más que un avance se ha producido una verdadera revolución. El descubrimiento de la informática, su aplicación paulatina en todo tipo de áreas de conocimiento y de producción, así como su introducción en el común de la población a través de todo tipo de componentes ha cambiado nuestra sociedad y nuestra economía más rápido que cualquier otro hecho o descubrimiento anterior.

El computador u ordenador es uno de los inventos que mejor resume la nueva situación tecnológica. Aparecieron primero como enormes y costosas máquinas que solamente estaban disponibles en importantes universidades o centros de investigación. Con la aparición de nuevas técnicas de fabricación, como los circuitos integrados, su tamaño, sus capacidades, y sobre todo precio, variaron de tal forma que se convirtieron en un producto de masas más, como lo podían ser la televisión o la radio. La aparición de Internet, y sobre todo su apertura al público general, determinaron de forma inequívoca la importancia de los ordenadores en la vida social, laboral o académica de cualquier persona hasta el día de hoy.

Simultáneamente a la aparición de Internet como servicio abierto, a principios de la década de 1990, surgió otro medio de comunicación que, si bien era más antiguo, se reinventaba a sí mismo gracias a los cambios en su tradicional soporte: la telefonía móvil. El *boom* en la implantación de Internet, junto al furor de la telefonía móvil, confirmaba que esta revolución tecnológica no sólo afectaba a la investigación o la actividad económica, sino que implicaba un fenómeno sociológico donde la comunicación y el acceso a la información en cualquier lugar y momento eran sus pilares básicos.

Como no podía ser de otra manera, la reducción del tamaño de los componentes y el aumento de sus prestaciones permitió acercar cada vez más ambos mundos, de forma que a través de un teléfono móvil no sólo se podían hacer llamadas o enviar SMS, sino que además se podía tener un acceso más o menos limitado a Internet, o incluso funciones añadidas como realizar fotografías o vídeos. Otros aparatos de similar tamaño, no directamente relacionados con la telefonía, surgieron y se hicieron tan populares como los primeros. Desde ese momento puede empezar a usarse el término genérico **dispositivo móvil**.

Así pues, un dispositivo móvil es un término general que describe una amplísima familia de aparatos electrónicos surgidos en los últimos años, de reducido tamaño, que ofrecen alguna capacidad de procesamiento y almacenamiento de datos y que están orientados a una función concreta o varias de ellas: desde los teléfonos móviles más evolucionados (los llamados *smartphones*), a ordenadores portátiles, cámaras digitales, reproductores de música o consolas de videojuegos.

La mayoría de estos aparatos cuentan con un sistema operativo de mayor o menor complejidad, que permita realizar las tareas de gestión de memoria y control de hardware que precisan. En el caso de los ordenadores portátiles, con tanta o incluso mayor capacidad que los de sobremesa, los sistemas operativos habituales son perfectamente compatibles y funcionan sin diferencias. Sin embargo, en otros dispositivos móviles es preciso diseñar nuevos sistemas operativos adaptados específicamente a sus características: restricciones de memoria y procesamiento, consumo mínimo de energía o gran estabilidad en su funcionamiento, entre otros.

Algunos sistemas operativos para dispositivos móviles más utilizados son Symbian, con más de un 60% del mercado, y Windows Mobile, la versión móvil del gigante Microsoft. Este reparto puede verse sustancialmente afectado por la incursión de otro gigante de Internet, Google, a través de su propio sistema operativo de nombre Android [1].

Google es una joven compañía surgida a finales de los 90 que pronto se hizo muy popular gracias al potente buscador del mismo nombre. Durante años, Google no ha dejado de crecer y de ofrecer toda clase de servicios basados siempre en Internet y en la combinación de la última tecnología disponible con la máxima experiencia del usuario.

Android, más que un sistema operativo, representa toda una pila de software para dispositivos móviles que incluye gran cantidad de drivers, gestor de bases de datos, una completa *framework* de aplicaciones, y numerosas aplicaciones de usuario. Android está basado en el núcleo de Linux y todas sus aplicaciones se escriben en lenguaje Java, disponiendo además de una máquina virtual específica llamada Dalvik.

Con la aparición de este sistema, Google pretende aprovechar al máximo la cada vez mayor capacidad de los dispositivos móviles, que llegan a incluir componentes como GPS, pantallas táctiles, conexiones rápidas a Internet, y por supuesto, todos los servicios asociados hasta ahora a los teléfonos móviles, además de aplicaciones de usuario hasta ahora limitadas a los ordenadores, como clientes de correo, aplicaciones ofimáticas o videojuegos. En Android, cada aplicación corre en su propio proceso, donde el sistema decide en cada momento qué aplicación debe ser eliminada para liberar recursos en caso de carencia, y responsabilizándose igualmente de restaurarla de forma totalmente transparente al usuario. Navegar entre varias aplicaciones abiertas deja de ser una característica propia de ordenadores.

Android se lanza bajo la licencia Apache, lo que implica que, como software libre, cualquier desarrollador tiene acceso completo al SDK del sistema, incluidas todas sus API, documentación y emulador para pruebas, pudiendo distribuirlo y modificarlo.

Además, esta licencia permite a los desarrolladores tanto publicar a su vez sus creaciones, como distribuirlas únicamente bajo pago ocultando el código fuente. Este nuevo sistema introduce también interesante conceptos, como es la composición de sus aplicaciones a través de combinación de módulos o bloques básicos, según la naturaleza de la aplicación, o la delegación en el sistema de determinadas acciones para que sean otras aplicaciones instaladas las que se hagan cargo de ellas.

En el momento de esta redacción, Android no acaba más que comenzar su andadura en el mercado de los sistemas operativos para dispositivos móviles. Únicamente existe un terminal a la venta, frente a los miles de modelos de sus competidores [2]. Numerosos fabricantes, distribuidores y operadores se han unido a la plataforma de patrocinadores de Android, y fabricantes de gran renombre han anunciado la producción inminente de nuevos modelos con Android como sistema nativo [3]. Dada la creciente importancia de estos dispositivos y su más que probable implantación masiva, cada vez con mayores prestaciones y capacidades, unida al éxito que suele acompañar a Google en sus proyectos, todo parece indicar que Android podría posicionarse en un futuro más o menos cercano como uno de los sistemas operativos más utilizados en el mundo.

## *1.2 Objetivos*

El nuevo sistema operativo para dispositivos móviles de Google, Android, centra el desarrollo de este proyecto fin de carrera. Para poder dirigir con mayor éxito los esfuerzos por conocer y comprender las características de este nuevo sistema, es necesario fijar unos objetivos que abarquen las actividades que se pretenden realizar y, además, permitan al final de las mismas conocer el grado de desarrollo y cumplimiento alcanzado.

Por ello, los objetivos perseguidos en el desarrollo de este proyecto fin de carrera son los enumerados a continuación:

- **Conocer las principales características de Android**. El primer paso para conocer este nuevo sistema debe consistir en indagar toda la información posible sobre él, a fin de conocer cuál es su arquitectura, sus componentes básicos, y cuál es su comportamiento al ejecutar las aplicaciones, documentando todos estos aspectos. Además, ha de averiguarse cuáles son las ventajas y las posibilidades reales que Android ofrece frente a otros sistemas de similar naturaleza.

- **Estudiar el entorno de desarrollo de Android**. Al lanzarse bajo una licencia de software libre, el SDK completo está disponible para cualquier desarrollador que desee descargarlo. Este incluye numerosas ayudas para comenzar a crear aplicaciones en Android, desde las API completas con todas las clases y paquetes, hasta herramientas de programación y un completo emulador para poder realizar pruebas. Todos estos elementos han de ser estudiados y explicados.

- **Desarrollar una aplicación completa para Android**. Una vez conocidas las características de este sistema, así como el entorno de desarrollo que ofrece y sus posibilidades, debe crearse una aplicación que aproveche algunas de sus características más fundamentales. A través de este desarrollo y una detallada documentación del mismo, el lector debe poder comprender mejor el funcionamiento de aplicaciones para Android, así como conocer los pasos para crear sus propias aplicaciones.

## *1.3 Contenido de la memoria*

A continuación se explica brevemente el contenido de cada capítulo y anexo incluido en esta memoria.

En el capítulo 1, *Introducción*, se expone la motivación del presente proyecto, los objetivos que persigue, así como una visión general de los contenidos de la memoria.

El capítulo 2, *Estado del Arte*, ofrece una descripción general de algunos de los aspectos técnicos relacionados con este proyecto, como los dispositivos móviles, los sistemas operativos más extendidos para estos, el lenguaje Java ME o un breve repaso de la trayectoria de Google.

En el capítulo 3, titulado *La Plataforma Android*, se explica al lector las características básicas del nuevo sistema operativo Android, su diseño, arquitectura y funcionamiento, así como una guía de instalación para el entorno de desarrollo Eclipse, finalizando con la implementación de un sencillo ejemplo "Hola Mundo".

El capítulo 4, de nombre *ContactMap: Localizador de Contactos*, desribe de forma detallada el completo desarrollo de una aplicación para Android: desde su diseño y funcionalidad, hasta la implementación de cada una de sus clases y componentes básicos, haciendo énfasis en la utilización de los aspectos más novedosos ofrecidos por Android.

El capítulo 5, *Historia del Proyecto*, ofrece al lector una breve explicación de los pormenores en la vida del proyecto, la evolución de su desarrollo en el tiempo y un presupuesto justificado.

El capítulo 6, *Conclusiones y Trabajos Futuros*, se exponen los resultados obtenidos tras la finalización del proyecto, comparándolos con los objetivos marcados inicialmente, y trazando posibles líneas futuras de desarrollo.

Por último, se han incluido una serie de anexos donde se puede encontrar el *ANEXO A: Manual de Instalación* de *ContactMap*, una descripción del CD adjunto a esta memoria en *ANEXO B: Contenido del CD*, así como la recopilación de la bibliografía y los conceptos utilizados a lo largo del proyecto en *ANEXO C: Referencias* y *ANEXO D: Términos*, respectivamente.

# 2 ESTADO DEL ARTE

## *2.1 Dispositivos móviles*

No existe un consenso claro a la hora de definir qué es realmente un dispositivo móvil y qué no lo es. Es frecuente que hoy en día este término se utilice para designar únicamente a ciertos modelos de teléfonos móviles con mayores o menores prestaciones. A pesar de ello, un dispositivo móvil no tiene por qué ceñirse solamente al ámbito telefónico.

Buscando ser más rigurosos, se podría denominar dispositivo móvil a todo aparato electrónico que cumple unas características muy básicas:

- es de reducido tamaño, haciéndolo fácil de transportar.
- cuenta con una cierta capacidad de computación y almacenamiento de datos.
- incorpora elementos de E/S básicos (por lo general, pantalla y/o algún tipo de teclado).

Más allá de estas características comunes, los dispositivos móviles forman en la actualidad un grupo sumamente heterogéneo y pueden incorporar casi cualquier componente de hardware y software que amplía y diversifica su función inicial. El más frecuente sin duda es la conexión telefónica (incluyendo servicios como el envío de SMS, MMS, y acceso WAP) o la conexión a Internet.

Igualmente son habituales la cámara fotográfica y de vídeo, pantalla táctil, teclado QWERTY, receptor de radio, Bluetooth, conexión mediante infrarrojos, dispositivos de memoria extraíbles, localizador GPS, acelerómetro, etc. Desde el punto de vista del software, pueden incorporar también un amplio abanico de aplicaciones tales como programas ofimáticos, reproductores de audio y vídeo, organizadores, videojuegos, navegadores web o clientes de correo, entre otros.

### 2.1.1 Clasificación de los dispositivos móviles

Al igual que ocurre a la hora de definir qué es un dispositivo móvil, la clasificación que se pueda hacer de estos aparatos está sujeta a diferentes valoraciones y a veces no existe un acuerdo amplio para ubicar un dispositivo móvil en una determinada familia.

En la década de los 90, tras la aparición de estos primeros dispositivos, establecer clasificaciones más o menos rigurosas era posible debido a que cada aparato estaba claramente definido para una función determinada o para un público concreto. El aumento de las prestaciones y funcionalidades que en la actualidad puede ofrecer cualquier dispositivo móvil dificulta el poder agruparlo dentro de un conjunto determinado. Por ejemplo, un *smartphone* representa una evolución de un teléfono móvil tradicional, esto es, su cometido es ofrecer comunicación telefónica; sin embargo,

cuenta con otros servicios adicionales como la conexión a Internet y aplicaciones, servicios propios de un ordenador, cámara de fotos y de vídeo o la posibilidad de reproducir películas o videojuegos.

La clasificación que a aquí se propone utilizar como principal criterio la funcionalidad o servicio principal para la que ha sido diseñado bien el propio dispositivo móvil, o bien aquel dispositivo del que directamente procede y del que supone una evolución mejorada. Tal y como se deduce de todo lo anteriormente explicado, la pertenencia a una categoría concreta no implica en absoluto que el dispositivo no pueda ofrecer otras muchas características propias de otras categorías.

Dicho la anterior, los dispositivos móviles pueden ser clasificados en los siguientes grupos:

- Dispositivo de comunicación

  Un dispositivo de comunicación es aquel dispositivo móvil cuyo cometido principal es ofrecer una infraestructura de comunicación, principalmente telefónica. Estos dispositivos ofrecen además servicios como el envío de mensajes SMS y MMS, o acceso WAP. En esta categoría se incluiría el tradicional teléfono móvil, precursor indiscutible dentro de los dispositivos móviles, la BlackBerry y el *smartphone*, que amplía considerablemente las prestaciones del primero mediante pantalla táctil, conexión a Internet o la ejecución de aplicaciones (por ejemplo, iPhone o HTC G1).

- Dispositivo de computación

  Los dispositivos de computación son aquellos dispositivos móviles que ofrecen mayores capacidades de procesamiento de datos y cuentan con una pantalla y teclado más cercanos a un ordenador de sobremesa. Dentro de este grupo encontramos a las PDA, muy populares a finales de los años 90 y que permitían al usuario disponer de un organizador mucho más completo que los ofrecidos por los teléfonos móviles del momento, e incluso en ocasiones la visualización de documentos o acceso a Internet. Por otro lado, dispositivo de computación también es un ordenador portátil o *laptop*, que dentro de los dispositivos móviles son sin duda los que mayores prestaciones hardware ofrecen (igualando o superando a los de sobremesa) pero también los que tienen, con diferencia, un mayor tamaño, peso y precio. Las calculadoras gráficas pueden ser igualmente incluidas en este grupo de dispositivos de computación.

- Reproductor multimedia

  Un reproductor multimedia es aquel dispositivo móvil que ha sido específicamente diseñado para proporcionar al usuario la reproducción de uno o varios formatos de datos digitales, ya sea audio, vídeo o imágenes. Dentro de estos dispositivos encontramos reproductores de MP3, los DVD portátiles, los eBooks, y en los últimos años los reproductores multimedia de la popular familia

iPod de Apple, que ofrecen tanto audio y como vídeo. Estos dispositivos son con frecuencia los de más reducido tamaño y, junto a los teléfonos móviles y *smartphones*, los más extendidos.

- Grabador multimedia

  Dentro de los dispositivos móviles, un grabador multimedia es aquel dispositivo que posibilita la grabación de datos en un determinado formato digital, principalmente de audio y vídeo. En esta categoría se hallan las cámaras fotográficas digitales o las cámaras de vídeo digital.

- Consola portátil

  Una consola portátil es un dispositivo móvil cuya única función es la de proporcionar al usuario una plataforma de juego. Las consolas portátiles fueron, junto a los teléfonos, los primeros dispositivos móviles en convertirse en un producto de masas. Hoy en día representan un importantísimo volumen de ventas dada su gran aceptación en la sociedad y son objeto de auténticas guerras comerciales entre las principales compañías del sector. Algunos ejemplos de esta categoría son la Nintendo DS de Nintendo, o la PSP de Sony.

En la siguiente figura, Figura 1, se ofrece una amplia muestra de los distintos dispositivos móviles que pueden ser encontrados hoy día en el mercado.


**Figura 1. Ejemplos de dispositivos móviles.**

a) teléfono móvil Nokia 3210; b) BlackBerry 8300; c) *smartphone* HTC G1; d) PDA Acer N35; e) ordenador portátil MacBook Air; f) ebook Sony Reader; g) reproductor iPod Nano; h) cámara de fotografía digital Nikkon Coolpix S210; i) consola portátil PSP; j) consola portátil Nintendo DS

### 2.1.2 Smartphone

Dentro de los dispositivos móviles, un *smartphone* (cuya traducción en español sería "teléfono inteligente") es una evolución del teléfono móvil tradicional que cuenta con ciertas características y prestaciones que lo acercan más a un ordenador personal que a un teléfono tradicional.

Entre dichas características, se puede encontrar una mejora en la capacidad de proceso y almacenamiento de datos, conexión a Internet mediante Wi-Fi, pantalla táctil, acelerómetro, posicionador geográfico, teclado QWERTY y diversas aplicaciones de usuario como navegador web, cliente de correo, aplicaciones ofimáticas, reproductores de vídeo y audio, etc. incluyendo la posibilidad de descargar e instalar otras nuevas.

A pesar de estas importantes mejoras con respecto a sus predecesores móviles, el reducido tamaño de los *smartphones* conlleva inexorablemente limitaciones de hardware que los mantienen claramente diferenciados de los ordenadores convencionales. Estas limitaciones se reflejan principalmente en pantallas más pequeñas, menor capacidad del procesador, restricciones de memoria RAM y memoria

persistente, y necesidad de adaptar el consumo de energía a la capacidad de una pequeña batería.

Estas limitaciones obligan a tener muy presente la capacidad real del dispositivo a la hora de desarrollar su software, ya sean aplicaciones de usuario o el propio sistema operativo.

## 2.2 Sistemas operativos para dispositivos móviles

El sistema operativo destinado a correr en un dispositivo móvil necesita ser fiable y tener una gran estabilidad, ya que incidencias habituales y toleradas en ordenadores personales como reinicios o caídas no tienen cabida en un dispositivo de estas características. Además, ha de adaptarse adecuadamente a las consabidas limitaciones de memoria y procesamiento de datos, proporcionando una ejecución exacta y excepcionalmente rápida al usuario.

Estos sistemas han de estar perfectamente testeados y libres de errores antes de incorporarse definitivamente a la línea de producción. Las posibilidades que existen en un ordenador estándar de realizar actualizaciones e incluso reinstalar mejores versiones del sistema para cubrir fallos o deficiencias son más limitadas en un dispositivo móvil.

Es posible incluso que un aparato de esta naturaleza deba estar funcionando ininterrumpidamente durante semanas e incluso meses antes de ser apagado y reiniciado, a diferencia de lo que ocurre con un ordenador personal. El consumo de energía es otro tema muy delicado: es importante que el sistema operativo haga un uso lo más racional y provechoso posible de la batería, ya que esta es limitada y el usuario siempre exige una mayor autonomía.

Todos estos aspectos de los dispositivos móviles, entre otros muchos, han de ser tenidos en cuenta a la hora de desarrollar un sistema operativo competente en el mercado, atractivo para los fabricantes y que permita al usuario sacar máximo provecho de su terminal [4].

En la actualidad, existen varios sistemas operativos para toda la gama de dispositivos móviles. Dentro de los *smartphones*, Symbian se lleva la mayor cuota de mercado con un 65%, seguido de Windows Mobile con un 12% [5].

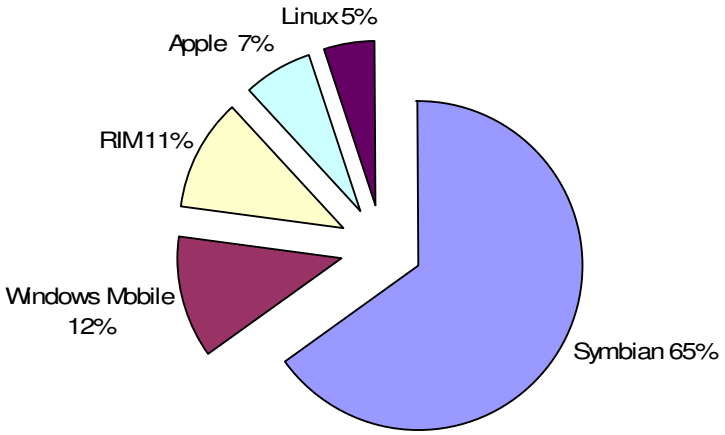


**Figura 2. Cuota de mercado de distintos SSOO para *smartphones* (2007)**

### 2.2.1 Symbian

Symbian es un sistema operativo para dispositivos móviles desarrollado por Psion, Nokia, Motorola y Ericsson. El principal objetivo de estas compañías era el de crear un nuevo y compartido sistema operativo que estuviera perfectamente adaptado a los teléfonos móviles del momento, y fuese además capaz de competir con Palm OS y Windows Mobile. La primera versión de Symbian, basada en el sistema EPOC de Psion, se lanzó en 1998. Actualmente, el número de empresas vinculadas al proyecto ha crecido considerablemente, siendo la última versión lanzada la 9.3 en julio de 2006.

El acuerdo bajo el cual se desarrolló Symbian es bastante simple: Symbian Ltd. desarrolla el sistema operativo Symbian, que incluye el microkernel, los controladores, el *middleware* y una considerable pila de protocolos de comunicación e interfaces de usuario muy básicas. Los desarrolladores que obtienen la licencia correspondiente para trabajar con Symbian implementan sus propias interfaces de usuario y conjuntos de aplicaciones según las necesidades de sus propios dispositivos. Esto permitió a Symbian posicionarse como un sistema operativo muy flexible, que tenía en cuenta los requisitos de la mayoría de los dispositivos fabricados y, al mismo tiempo, permitía un alto grado de diferenciación [6].

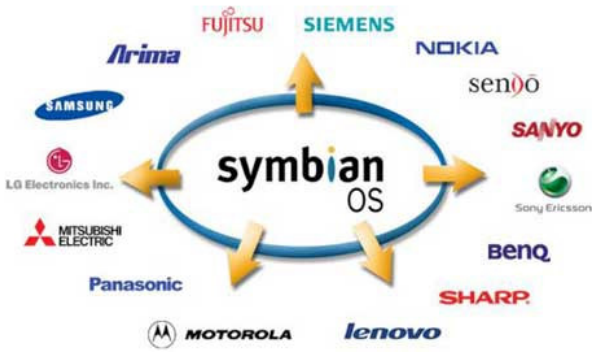


**Figura 3. Entidades vinculadas a Symbian**

En Symbian, una mínima porción del sistema tiene privilegios de kernel; el resto se ejecuta con privilegios de usuario en modo de servidores, de forma que los procesos en ejecución y sus prioridades son manejadas por este microkernel. Cada una de las aplicaciones corre en su propio proceso y tiene acceso únicamente a una exclusiva zona de memoria.

Symbian contempla cinco tipos de ediciones o series del sistema operativo según las características del dispositivo móvil [7]. La principal diferencia entre ediciones no radica tanto en el núcleo del sistema operativo como en la interfaz gráfica utilizada:

- **Serie60**. El más popular de todos, debido fundamentalmente a que el gigante Nokia, uno de los fabricantes más importantes del mundo, ha hecho de Symbian y de su versión Serie60 el núcleo de casi todos sus modelos de *smartphones*. Los dispositivos con Serie60 tiene una pantalla pequeña y un teclado del tipo 0-9#. También lo utilizan fabricantes como Siemens, Samsung y Panasonic.

- **Serie80**. Esta edición, también usada por Nokia, está más orientada a dispositivos que tienen pantalla táctil y permiten multitarea, pudiendo tener varias aplicaciones abiertas simultáneamente.

- **Serie90**. Muy similar a la edición Serie80, sólo que éstos dispositivos tienen una pantalla más grande y llevan incorporados sensores táctiles más desarrollados. Utilizan teclados virtuales, reconocimiento de trazos o teclados acoplables mediante, por ejemplo, Bluetooth.

- **UIQ**. La interfaz de esta edición de Symbian se encuentra muy influenciada por Palm OS. Implementan una especie de multitarea virtual, dando al usuario la falsa sensación de poder realizar varias acciones simultáneas; suelen tener un alto coste computacional e influyen negativamente en el tiempo de respuesta apreciado por el usuario. Es utilizado en algunos modelos de Sony Ericcson y Motorola.

- **MOAP**. Esta edición se da únicamente en Japón, principalmente en el fabricante FOMA.

Para más información acerca de las distintas ediciones y los modelos de dispositivos donde actualmente funcionan puede consultarse la referencia [8].

Desarrollar aplicaciones para Symbian es relativamente sencillo. No es necesario aprender ningún lenguaje de programación nuevo porque permite utilizar lenguajes habituales como Java, C++, Visual Basic o Perl, entre otros, para desarrollar aplicaciones. Este hecho ha permitido que actualmente sean cientos de miles las aplicaciones y utilidades disponibles para Symbian.

### 2.2.2 Windows Mobile

Windows Mobile es un sistema operativo diseñado por Microsoft y orientado a una gran variedad de dispositivos móviles. En realidad, Windows Mobile representa una particularización de otro gran sistema de Microsoft llamado Windows CE.

A principios de la década de los 90, cuando comenzaron a aparecer los primeros dispositivos móviles, Microsoft tomó la decisión de crear un sistema operativo capaz de hacer frente al entonces recientemente lanzado por Apple, el sistema Newton MessagePad. Fruto de esta iniciativa surgió Pegasus, cuyo nombre comercial definitivo fue Windows Compact Edition, o Windows CE [9].

El objetivo principal que buscaba Microsoft era que el nuevo sistema fuera lo suficientemente flexible y adaptable para poder ser utilizados en un amplio abanico de dispositivos, cuyo única característica común es la de ser de reducido tamaño y tener, por tanto, una limitación obvia en sus recursos. Las características principales con las que cuenta Windows CE son las siguientes [10]:

- Es un sistema modular, lo que permite que cada fabricante pueda seleccionar aquellas partes que le benefician más para su dispositivo.
- Contempla una considerable gama de recursos hardware: teclado, cámara, pantalla táctil, etc.
- Tiene un tamaño en memoria relativamente pequeño y bajo coste computacional.
- Es capaz de trabajar con distintas familias de procesadores de 32 bits.
- Permite interactuar con otros dispositivos móviles.

Un aspecto distintivo de Windows CE con respecto a otros productos desarrollados por Microsoft es que un elevado número de sus componentes se ofrece a los fabricantes y desarrolladores a través del propio código fuente. Esto les permite poder adaptar el sistema a sus dispositivos específicos. Aquellos componentes básicos de Windows CE que no necesitan ningún tipo de adaptación siguen siendo ofrecidos únicamente como código binario. La arquitectura básica de Windows CE es la explicada a continuación [11]:

- **OEM Layer**: es la capa situada entre el hardware del dispositivo y el kernel. Permite a los fabricantes desarrollar sus propios drivers y funciones de control de los elementos de hardware.

- **Operating System Layer**: incluye el kernel como elemento principal y el conjunto de API Win32 necesarias. En esta capa se sitúan los bibliotecas de comunicaciones, el gestor gráfico, gestor de ficheros y registros, así como otros componentes opcionales.

- **Application Layer**: donde residen las aplicaciones por defecto de Windows CE y las aplicaciones del usuario.

Actualmente, Windows CE en su versión 6.0, es una gran colección de módulos que permiten construir un completo sistema operativo, permitiendo así seleccionar y configurar aquellos módulos que son realmente necesarios para un determinado dispositivo o una aplicación. En esta capacidad de adaptación es donde surge en el año 2003 Windows Mobile, una especificación de ciertas partes de Windows CE adaptadas especialmente a *smartphones* y PDA. Windows Mobile está a su vez dividido en tres ediciones [12]:

- **Windows Mobile Classic**: antes llamado Pocket PC, orientado a dispositivos del tipo PDA sin funciones de comunicación telefónica.

- **Windows Mobile Standard**: hasta ahora conocido como Smartphone, esta edición está destinado a dispositivos del tipo *smartphone* sin pantalla táctil.

- **Windows Mobile Professional**: anteriormente Pocket PC Phone Edition, para cubrir aquellas PDA u ordenadores de bolsillo con capacidad de comunicación telefónica.

## 2.3 Java 2 Micro Edition

Java es un lenguaje de programación orientado a objetos creado por Sun Microsystems. Su principal aliciente, y la característica que lo convirtió en uno de los lenguajes más populares hasta hoy, fue la posibilidad de crear aplicaciones independientes a la plataforma donde van a ser ejecutadas, siguiendo el axioma "*write once, run anywhere"* ("escribe una vez, ejecuta en cualquier parte"). Java ofrece la oportunidad de crear aplicaciones tanto de escritorio como empresariales, incluyendo por supuesto a los dispositivos móviles [13].

Esta portabilidad, que lo hizo tan importante, viene dada por la máquina virtual de Java o JVM (Java Virtual Machine). Al compilar un fichero fuente en Java no se crea un binario directamente ejecutable, sino un código intermedio llamado *bytecode* en ficheros cuya extensión es .class. Para cada hardware debe haber una JVM específica, ya sea un teléfono móvil, un ordenador con Windows, o un microondas. Cada máquina

virtual conoce el conjunto de instrucciones de la plataforma sobre la que está instalada, y puede traducir el *bytecode*, común para todas las máquinas, al código nativo que es capaz de entender el hardware en cuestión de dicha plataforma.

### 2.3.1 Ediciones de Java 2

La revolución tecnológica acaecida en el los últimos años propiciaron que los responsables de Java ofrecieran soluciones personalizadas a cada ámbito tecnológico. Sun decidió crear una edición distinta de Java según las necesidades del entorno y la tecnología utilizada:

- Java Enterprise Edition (Java EE), orientada al entorno empresarial.
- Java Standard Edition (Java SE), orientada al desarrollo con independencia de la plataforma.
- Java Micro Edition (Java ME), orientada a dispositivos con capacidades restringidas.
- Java Card, orientada a tarjetas inteligentes o *smart cards*.

Todas las ediciones comparten un conjunto más o menos amplio de las API básicas de Java, agrupadas principalmente en los paquetes `java.lang` y `java.io`. A día de hoy, Java EE representa un superconjunto de Java SE, pues contiene toda la funcionalidad de éste y más características, así como Java ME es un subconjunto de Java SE (excepto por el paquete `javax.microedition`).

Por lo tanto, Java Micro Edition es una versión muy específica del lenguaje Java, creado para desarrollar, instalar y ejecutar software escrito en Java en aparatos electrónicos de baja capacidad, como electrodomésticos, PDA, teléfonos móviles u ordenadores de bolsillo [14].

### 2.3.2 Arquitectura de Java ME

Una aplicación Java ME típica está formada por un archivo JAR, que es el que contiene a la aplicación en sí, y un archivo opcional JAD (Java Archive Descriptor) que contiene diversa información sobre la aplicación.

Antes de desarrollar una aplicación en Java ME, es necesario tomar una serie de decisiones según el dispositivo a utilizar y las necesidades requeridas. No todos los desarrollos realizados en Java ME utilizan los mismos componentes. En concreto, una aplicación en Java ME se desarrolla a partir de una combinación de:

- Máquina virtual: existen disponibles dos máquinas virtuales de Java ME con diferentes requisitos, cada una pensada para tipos distintos de pequeños dispositivos:

    - KVM, o Kilobyte Virtual Machine, se corresponde con la máquina virtual más pequeña desarrollada por Sun. Se trata de una implementación de máquina virtual reducida y orientada a dispositivos de 16 o 32 bits con al menos 25 Mhz de velocidad y hasta 512 Kb de memoria total disponible.

    - CVM, o Compact Virtual Machine, soporta las mismas características que la Máquina Virtual de Java SE. Está orientada a dispositivos electrónicos con procesadores de 32 bits de gama alta y en torno a 2Mb o más de memoria RAM.

- Configuración: una configuración consiste en un conjunto de clases básicas destinadas a conformar el corazón de la aplicación. En concreto, dentro de Java ME existe dos configuraciones:

    - Connected Limited Device Configuration (CLDC) enfocada a dispositivos con restricciones de procesamiento y memoria.

    - Connected Device Configuration (CDC) enfocada a dispositivos con más recursos.

- Perfil: bibliotecas de clases específicas, orientadas a implementar funcionalidades de más alto nivel para familias específicas de dispositivos.

Cada una de las dos configuraciones mencionadas requiere en realidad de una determinada máquina virtual. De esta forma, si es escoge la configuración CLDC, será necesario utiliza la máquina virtual denominada CVM; si, por el contrario, se decide utilizar la configuración CDC, la máquina virtual necesaria es la conocida como KVM.

Con la elección de perfiles, se da una situación similar. Existen unos perfiles que se utilizan sobre la configuración CDC (Foundation Profile, Personal Profile y RMI Profile) y otros que lo hacen sobre CLDC (PDA Profile y Mobile Information Device Profile, conocido como MIDP).

Las aplicaciones en Java ME que se realizan utilizando el perfil MIDP reciben el nombre de MIDlets. Se dice así que un MIDlet es una aplicación Java ME realizada con el perfil MIDP, sobre la configuración CLDC, y usando la máquina virtual KVM.

Desde un punto de vista práctico, MIDP es el único perfil actualmente disponible.

### 2.3.3 Bibliotecas disponibles

Al desarrollar una aplicación en Java ME bajo el perfil MIDP, el programador puede hacer uso, entro otras, de las siguientes bibliotecas:

| Nombre | Descripción |
|---|---|
| `java.io` | Operaciones de E/S básicas |
| `java.lang` | Operaciones de la Máquina Virtual |
| `java.util` | Utilidades estándar |
| `javax.microedition.midlet` | Marco de ejecución para las aplicaciones |
| `javax.microedition.lcdui` | Interfaces de usuario |
| `javax.microedition.rms` | Almacenamiento persistente en el dispositivo |
| `javax.microedition.io` | Conexión genérica |

**Tabla 1. Bibliotecas para MIDP de Java ME**

A través de la configuración CLDC, se obtienen un conjunto de clases heredadas de la edición Java SE. Aproximadamente, unas 37 clases derivadas de los paquetes `java.lang`, `java.util` y `java.io`. Cada clase es un subconjunto de la correspondiente en Java SE.

Los paquete `java.io` y `java.net`, son los encargados en Java SE de las operaciones de E/S. Debido a las limitaciones de CLDC, no fue posible incluir todas y cada una de las clases de estos paquetes. Tampoco se incluyeron la clase `java.net`, encargada de las comunicaciones TCP/IP. En su lugar, MIDP añade el paquete `javax.microedtion.io` que suple de forma genérica las necesidades en cuestiones de comunicación, como se verá más adelante.

Además de los ya mencionados, MIDP incluye también un paquete que posibilita la creación de interfaces de usuario: `javax.microedition.lcdui`. Las clases contenidas en este paquete pueden dividirse en dos grande grupos:

- Elementos de interfaz de usuario de alto nivel. Esta interfaz usa componentes tales como botones, cajas de texto, formularios, etc. Estos elementos son implementados por cada dispositivo y la finalidad de usar las APIs de alto nivel es su portabilidad.

- Elementos de interfaz de usuario de bajo nivel. Se tiene un control total sobre los recursos, pero se puede perder la portabilidad ya que dependerá mucho del dispositivo destino. Generalmente, se utiliza en juegos.

El paquete `javax.microedition.rms` proporciona un mecanismo a los MIDlets para poder almacenar información de forma persistente en el dispositivo, utilizando para ello una base de datos basada en registros llamada Record Management System o RMS. Los registros se convierten así en la unidad básica de información en un MIDlet, permitiendo al dispositivo crear, leer, modificar y borrar cualquier registro de datos.

### 2.3.4 Comunicaciones en MIDlets

Anteriormente se ha mencionado que el paquete `javax.microedition.io` contiene las clases que dan soporte a las comunicaciones en una aplicación MIDP. De hecho, este paquete pretende sustituir, de una forma más breve y concisa, al paquete java.net de Java SE. Sin embargo, en una comunicación el MIDlet no utiliza únicamente este paquete, sino que también usará java.io, añadido por la configuración CLDC.

Mientras que el paquete `javax.microedtion.io` tiene como misión crear y manejar diferentes conexiones de red (por HTTP, datagramas, *sockets*), el paquete `java.io` lo que posibilita a través de su declaración de clases es poder leer y escribir en estas conexiones de red.

Debido a las limitaciones ya conocidas de un dispositivo basado en MIDP, es del todo imposible contener en él la implementación de cada protocolo de comunicación existente. Lo que proporciona `javax.microedition.io` es una única clase, llamada `Connector`, que esconde todos los detalles de la conexión. Dicho en otras palabras, una misma clase `Connector` sirve tanto para leer archivos, como abrir una conexión por HTTP o mediante *sockets*.

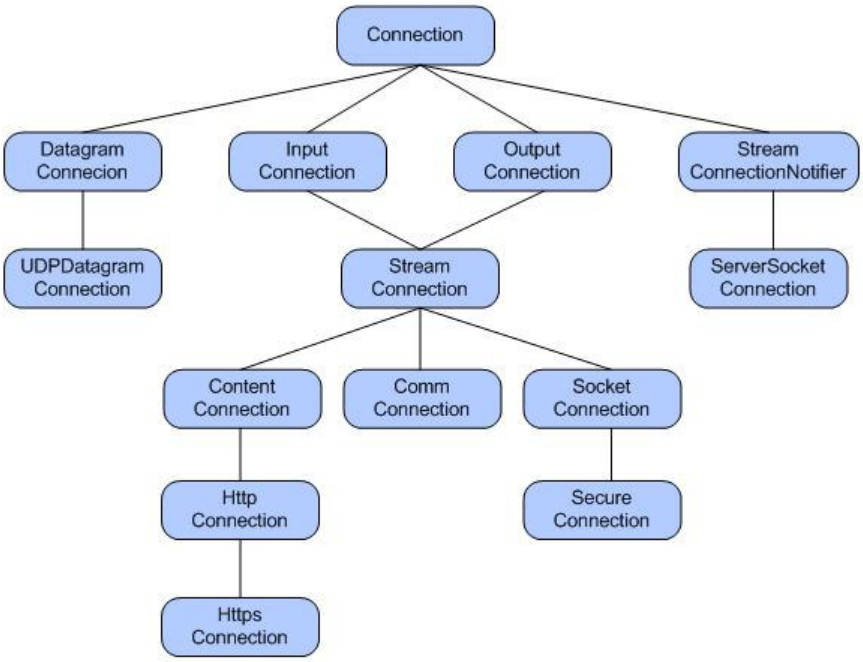


**Figura 4. Clases del paquete `javax.microedition.io`**

Se comentarán a continuación el cometido de algunas de las clases de comunicaciones mostradas en la Figura 4:

- `Connection`: representa la conexión en una forma genérica y abstracta.
- `InputConnection`: representa una conexión basada en *streams* de entrada.
- `OutputConnection`: representa una conexión basada en *streams* de salida.
- `StreamConnection`: representa un tipo de conexión cuyos datos pueden ser tratados como *streams* de bytes y en la que es posible leer y escribir.
- `DatagramConnection`: define las capacidades que debe tener una conexión basada en datagramas.
- `HttpConnection`: que implementa el protocolo HTTP.
- `UDPDatagramConnection`: representa una conexión basada en datagramas en los que se conoce su dirección final.
- `CommConnection`: representa una conexión donde los bits de datos se transmiten secuencialmente, es decir, en serie.

- `SocketConnection`: define una conexión entre *sockets* basados en *streams*.

### 2.3.5 El problema de la fragmentación

La popularidad de la tecnología Java hizo que pronto todos los fabricantes y desarrolladores se interesaran en su versión para dispositivos móviles. La creciente demanda de estos aparatos impulsó el uso de Java ME para el desarrollo de todo tipo de aplicaciones y juegos.

Sin embargo, en muchos casos las aplicaciones desarrolladas en Java ME no pudieron seguir la filosofía tradicional de Java "*write once run everywhere"* ("escribe una vez, ejecuta en cualquier parte"): con frecuencia, cada aplicación debía ser adaptada según el dispositivo móvil al que estuviera destinada. Es decir, una misma aplicación requería varios desarrollos para poder llegar al mayor número de usuarios posible.

La diferenciación, muy útil para los fabricantes por permitirles adaptar sus productos a las distintas necesidades de los usuarios, se convirtió en el mayor problema hasta el día de hoy para los desarrolladores de aplicaciones móviles. Este problema recibe el nombre de fragmentación [15]. La fragmentación aumenta los costes y el tiempo de desarrollo de una aplicación, que necesita llegar a cuántos más usuarios posibles para poder maximizar su rentabilidad.

Para comprender de dónde viene la fragmentación en Java ME y por qué no existe en otras versiones de Java, es necesario tener en cuenta distintos factores vinculados a los dispositivos móviles.

Para empezar, **el hardware** presente en estos aparatos es mucho más heterogéneo del que puede darse en los computadores convencionales, como ordenadores de sobremesa o portátiles. Las capacidades del procesador o de memoria pueden ser tan distintas que marquen la diferencia entre poder o no poder ejecutar una aplicación o un videojuego. También la existencia de muchos tipos de pantallas, con diferentes resoluciones y formas, sobrepasa la mera diferenciación en el número de pulgadas que existe, por ejemplo, entre los ordenadores convencionales.

Por otro lado, está el **sistema operativo**. A diferencia de los computadores tradicionales, no existe en los dispositivos móviles un sistema operativo que domine abrumadoramente el mercado. La razón es que cada fabricante crea su propio sistema operativo (o lo adapta como en el caso de Symbian, anteriormente visto) ya que éste viene determinado por el tipo de hardware al que tendrá que dar soporte, muy distinto entre diferentes modelos. Aunque la JVM tiene como misión abstraer a la aplicación escrita en Java ME de estos detalles, a la hora de la verdad solamente estarán disponibles si el sistema operativo contempla las API necesarias para controlar todos esos recursos hardware.

Además de lo anterior, también se encuentran las **API específicas** creadas por cada fabricante. Estas API, que permiten utilizar cámaras, Bluetooth, MMS, listados de

contactos, etc., posibilitan hacer aplicaciones mucho más sofisticadas, pero fuera del estándar MIDP especificado por Java. Muchos de estos añadidos están disponibles solamente en algunos modelos de dispositivos. Algunas compañías implementan **su propia JVM**, cuya misión principal es abstraer lo más posible del sistema operativo nativo, pero los criterios de implementación y seguimiento de los estándares de Java son bastante abiertos y las JVM resultantes suelen ser a menudo diferentes entre sí.

En resumidas cuentas, Java ME ha sido durante estos años la indiscutible tecnología bajo la cual se ha desarrollado multitud de aplicaciones de toda naturaleza para dispositivos móviles, especialmente los teléfonos móviles y los primeros *smartphones*. Sin embargo, las diferentes implementaciones de hardware, API y JVM de los fabricantes ha frustrado la posibilidad de desarrollar aplicaciones verdaderamente portables en Java ME, obligando a que cada creación esté siempre dirigida a una determinada familia de dispositivos y a realizar labores de traducción para poder cubrir mayores cuotas de mercado.

## *2.4 Historia de Google Inc.*

El nacimiento y desarrollo de Google Inc. es una de esas tantas historias que parecen cumplir a rajatabla los estereotipos del "sueño americano": una simple pero brillante idea que, con esfuerzo, talento y dedicación, se convierte desde cero en una realidad de éxito.

### 2.4.1 Origen del buscador Google

En 1995 dos estudiantes de la Universidad de Standford, Larry Page y Sergey Brin, descubren su común interés en averiguar alguna forma de poder conocer el nivel de importancia de una determinada página web [16]. En concreto, Larry Page trabajaba en una idea según la cuál una página tenía mayor o menor importancia en la red en función del número de páginas que la enlazaran a través de links. La idea derivaba del mundo universitario en donde Page se movía, en el que la relevancia de un artículo académico venía dada por las veces que era citado a su vez en el trabajo de otros investigadores.

De este modo, Page creó un primer prototipo llamado BackRub, que rastreaba los sitios web y almacenaba los enlaces hallados en ellas en una base de datos. Cuando comenzó a realizar pruebas con BackRub, Page consiguió determinar cuántos enlaces había entre las páginas, pero no hallaba la forma de clasificarlas en importancia, ya que la compleja telaraña de enlaces que se formaba implicaba un considerable cálculo recursivo. Es en este punto donde entra Sergey Brin, licenciado en Informática y Ciencias Matemáticas, que andaba buscando un tema interesante para su tesis doctoral. Brin siempre había destacado por su habilidad matemática y se puso a trabajar junto con Page en elaborar un algoritmo que pudiera clasificar la importancia de los enlaces.

Fruto de esta colaboración nació PageRank, la evolución del anterior proyecto de Page, que además de buscar y almacenar enlaces era capaz de clasificar la importancia de una

determinada página en función de éstos. Pronto se percataron de que PageRank no sólo buscaba enlaces, los almacenaba y clasificaba, sino que también permitía realizar búsquedas entre todos esos enlaces. De hecho, los resultados obtenidos eran mejores que aquellos dados por los buscadores más populares de la época, como por ejemplo Altavista. Además, PageRank era totalmente escalable: a mayor número de enlaces en su base de datos, mejor y más precisas eran las búsquedas que realizaba en ellos.

Viendo los buenos resultados, Page y Brin bautizaron de nuevo su creación con el definitivo nombre de Google. El extraño nombre proviene del término científico "googol", que denota al número 10 elevado a 100. En 1997, publicaron su buscador en un servidor web de la Universidad de Standford, convirtiéndose en muy poco tiempo en una de las herramientas web favoritas de los alumnos de esta universidad. Este éxito animó a sus creadores a continuar desarrollándolo y ampliándolo, comprando discos duros y CPU de segunda mano a sus propios compañeros e instalando sus servidores en la misma habitación de Sergey Brin. En pocos meses, Google comenzó a saturar la red de la Universidad de Standford, una de las de mayor capacidad del mundo en aquella época, debido al tráfico de consultas que recibía. Es entonces cuando deciden adquirir el dominio "google.com".

Las necesidades de hardware y espacio de Google no dejaban de crecer, hasta el punto de que Larry Page y Sergi Brian se ven incapaces de financiarlo, a pesar de las labores de intermediación de la propia universidad para vender el proyecto. Finalmente, en 1998 y tras una charla informal de 10 minutos con Andy Bechtolsheim (coofundador de Sun Microsystems y vicepresidente de Cysco Systems), éste les extiende un cheque por valor de 100.000 dólares para que paguen "sus discos duros y a los dichosos abogados, para los temas legales" (sic) [16]. Para poder hacer efectivo el cobro se ven obligados a fundar sobre la marcha Google Inc., ya que fue el nombre que escribió Bechtolsheim en el cheque.

Desde entonces, Google no ha parado de indexar páginas y se convirtió de forma gradual en el buscador consultado por la abrumadora mayoría de los usuarios de Internet. Larry Page y Sergi Brian no invirtieron ni un solo dólar en publicidad: el boca a boca y sus resultados fueron suficientes para darlo a conocer al mundo. Actualmente, tiene una media de 200 millones de consultas al día.

En 2008, Google Inc. recibió el Premio Príncipe de Asturias de Comunicación y Humanidades, por "favorecer el acceso generalizado al conocimiento" y su "contribución decisiva al progreso de los pueblos, por encima de fronteras ideológicas, económicas, lingüísticas o raciales" [17].

### 2.4.2 Otros productos de Google Inc.

Sin lugar a dudas, el producto estrella de Google Inc. es su potente buscador web. Hoy en día el uso de Internet sería muy diferente sin la posibilidad de realizar búsquedas con la precisión y rapidez que Google puede ofrecer.

Sin embargo, tras el espectacular crecimiento de Google y su primer servicio comercial, AdWords (que permitía asociar publicidad a determinadas búsquedas), Google Inc. no ha dejado de innovar. En pocos años ha ido creando nuevos servicios que, en la mayoría de los casos, han adquirido tal popularidad entre la gente, que para mucho usuarios sería difícil imaginar el uso cotidiano de Internet si no existieran. En todo ellos, Google Inc. parece querer imprimir la misma filosofía de empresa: un servicio web, de acceso libre, gratuito en su mayor parte, con una interfaz minimalista pero funcional, y orientado siempre a mejorar la experiencia del usuario y a la interconexión con otros servicios en red.

Entre los productos más populares a través de la Web que actualmente ofrece Google, podemos encontrar los siguientes [18]:

- **Gmail**

  Probablemente, el servicio más utilizado de Google Inc. junto con su buscador. Gmail es un completo servicio de correo electrónico lanzado en 2004, que pronto destacó por la gran capacidad de almacenamiento (actualmente más de 7 GB), tamaño de archivos adjuntos, facilidad para realizar búsquedas entre el historial, su formato de etiquetado y clasificación de mensajes, configuración de idiomas y la sencilla interfaz que ofrecía. Actualmente, se encuentra aún en versión beta e incorpora continuas mejoras, como el popular chat entre contactos con Gmail Chat.

- **Google Maps**

  Google Maps es un servicio de localización a través de la Web, que ofrece una vista aérea de casi cualquier lugar del mundo. Mediante este servicio, el usuario puede cambiar entre el punto de vista de una fotografía real de satélite o el de mapa. Además, permite navegar libremente por el plano, alejando y acercando la imagen hasta un nivel de precisión considerable. Ofrece también un servicio de mejor ruta entre dos puntos y de localización exacta mediante la introducción de una dirección cualquiera del globo. Recientemente ha sido mejorado con Google Street View, que añade una nueva perspectiva panorámica de 360º desde pie de calle.

- **Google Calendar**

  Google Calendar es un servicio web que ofrece una completa agenda y planificación de horarios. Cada usuario dispone de calendarios mensuales o semanales, diferenciados por colores según la actividad, con la posibilidad de detallar cada día de la semana y de programar alertas dirigidas a un teléfono móvil o un correo electrónico. Los usuarios pueden compartir entre sí calendarios y actividades.

- **Google Documents**

  Representa uno de los proyectos más ambiciosos que actualmente desarrolla Google Inc. El objetivo es crear una completa plataforma de herramientas ofimáticas utilizando únicamente la Web, sin necesidad de instalar ningún software. Pensado para poder competir algún día con el extendido Microsoft Office, Google Documents contempla actualmente un procesador de texto, hojas de cálculo y presentaciones.

- **Google Chrome**

  Google Chrome es el navegador web desarrollado por Goolge Inc. Es un proyecto de software libre y está basado en componentes de otros navegadores de código abierto, como WebKit y Mozilla. Su principal objetivo es ofrecer una navegación más rápida, estable y segura, sumado a una interfaz simple y eficiente. La versión beta de Google Chrome fue lanzada en septiembre de 2008 y aunque recibió una buena acogida, todavía presentaba algunas limitaciones que no le permiten competir directamente con Explorer de Microsoft o con Mozilla Firefox.

- **Youtube**

  Aunque no es un producto desarrollado por Google Inc., fue adquirido por éste en 2006. Youtube es un sitio web que permite a los usuarios subir y visualizar toda clase de vídeos. Además, los vídeos pueden recibir comentarios, votaciones, y los usuarios pueden crear sus propios canales de distribución. Su popularidad es tan grande que actualmente está entre las páginas más visitadas de todo Internet y se ha convertido en un auténtico fenómeno sociológico. Recientemente, ha debido suprimir gran parte de sus contenidos debido a la vulneración de los derechos de autor.

- **Picasa**

  Picasa es otra platafoma software no creada inicialmente por Google Inc., pero sí adquiridad en 2004. Este software permite organizar, editar y buscar las fotografías de un usuario. También se encuentra en desarrollo una nueva funcionalidad que permita reconocer imágenes para poder asociarlas a determinadas palabras o claves de búsqueda. A diferencia de otros servicios, Picasa sí necesita ser instalado en el equipo del usuario, aunque tiene algunas funcionalidades web.

- **Blogger**

  Adquirido por Google Inc. en 2003, Blogger es actualmente uno de los portales de blogs más populares en todo el mundo. Su facilidad de uso permite al usuario registrarse, crear un blog e introducir nuevas entradas en tan sólo unos pequeños pasos.

### 2.4.3 Críticas a Google Inc.

A lo largo de su corta pero fulgurante trayectoria, Google Inc. ha ido cosechando numerosos éxitos que la han convertido en la empresa de referencia dentro de las tecnologías de Internet. Para muchos, Google Inc. es esa empresa cuyos directivos visten vaqueros y camiseta, apenas rondan los 35 años y convierten en oro todo lo que tocan, generando grandes beneficios. Sin embargo, en los últimos años se han empezado a oír voces que critican algunos de sus comportamientos, y que cuestionan su popular eslogan "*Don't be evil*" ("No seas malvado").

Quizás el más sonado y que peor repercusión tuvo para la imagen de Google Inc. fue su incursión en China [19]. Para que el gobierno chino permitiera operar a la compañía en su territorio, Google Inc. se vio obligado a aceptar y colaborar con las férreas normas de censura de aquel país, bloqueando el acceso a multitud de páginas y servicios que eran considerados maliciosos para el pueblo chino o contrarios a los intereses de su régimen político.

Por otro lado, el nacimiento de Gmail no estuvo tampoco exento de críticas. El hecho de que los robots de la compañía recorriesen los mensajes de los usuarios en busca de palabras a las que asociar anuncios de publicidad, o las condiciones de uso que advertían del mantenimiento de mensajes aún después de haber sido borrados, puso en alerta a muchas asociaciones de consumidores que acusaban a Google Inc. de vulnerar la intimidad de las personas [20].

Además, Google Inc. ha recibido acusaciones de atentar contra la propiedad intelectual. En 2008 fue finalmente absuelto por reproducir en los resultados de sus búsquedas parte de los contenidos de algunas páginas web [21], mientras que ese mismo año fue requerido judicialmente a proporcionar los datos de los usuarios de Youtube al gigante mediático Viacom, para que así éste pudiera presentar una demanda por utilizar parte de sus producciones sin autorización [22].

# 3 LA PLATAFORMA ANDROID

Android constituye una pila de software pensada especialmente para dispositivos móviles y que incluye tanto un sistema operativo, como *middleware* y diversas aplicaciones de usuario. Representa la primera incursión seria de Google en el mercado móvil y nace con la pretensión de extender su filosofía a dicho sector.

Todas las aplicaciones para Android se programan en lenguaje Java y son ejecutadas en una máquina virtual especialmente diseñada para esta plataforma, que ha sido bautizada con el nombre de Dalvik. El núcleo de Android está basado en Linux 2.6.

La licencia de distribución elegida para Android ha sido Apache 2.0 [23], lo que lo convierte en software de libre distribución. A los desarrolladores se les proporciona de forma gratuita un SDK y la opción de un *plug-in* para el entorno de desarrollo Eclipse varios que incluyen todas las API necesarias para la creación de aplicaciones, así como un emulador integrado para su ejecución. Existe además disponible una amplia documentación de respaldo para este SDK [24].

El proyecto Android está capitaneado por Google y un conglomerado de otras empresas tecnológicas agrupadas bajo el nombre de *Open Handset Alliance* (OHA) [25]. El objetivo principal de esta alianza empresarial (que incluye a fabricantes de dispositivos y operadores, con firmas tan relevantes como Samsung, LG, Telefónica, Intel o Texas Instruments, entre otras muchas) es el desarrollo de estándares abiertos para la telefonía móvil como medida para incentivar su desarrollo y para mejorar la experiencia del usuario. La plataforma Android constituye su primera contribución en este sentido.

Cuando en noviembre de 2007 Google anunció su irrupción en el mundo de la telefonía móvil a través de Android [26], muchos medios especializados catalogaron este novedoso producto como un nuevo sistema operativo, libre y específico para teléfonos móviles. Sin embargo, los responsables del proyecto se han esforzado desde entonces en destacar que la motivación de Android es convertirse en algo más que un simple sistema operativo.

Con Android se busca reunir en una misma plataforma todos los elementos necesarios que permitan al desarrollador controlar y aprovechar al máximo cualquier funcionalidad ofrecida por un dispositivo móvil (llamadas, mensajes de texto, cámara, agenda de contactos, conexión Wi-Fi, Bluetooth, aplicaciones ofimáticas, videojuegos, etc.), así como poder crear aplicaciones que sean verdaderamente portables, reutilizables y de rápido desarrollo. En otras palabras, Android quiere mejorar y estandarizar el desarrollo de aplicaciones para cualquier dispositivo móvil y, por ende, acabar con la perjudicial fragmentación existente hoy día.

Además de todo ello, otro aspecto básico para entender la aparición de Android es que pretende facilitar la integración de estos dispositivos con las posibilidades cada día mayores ofrecidas por la Web. Por ejemplo, una aplicación desarrollada en Android

podría ser aquella que indicase al usuario, a través de Google Maps, la localización de sus diferentes contactos de la agenda y que avisase cuando éstos se encuentren a una distancia cercana o en una ubicación determinada.

Mejorar el desarrollo y enriquecer la experiencia del usuario se convierte, por tanto, en la gran filosofía de Android y en su principal objetivo.

## 3.1 Arquitectura

Como ya se ha mencionado, Android es una plataforma para dispositivos móviles que contiene una pila de software donde se incluye un sistema operativo, *middleware* y aplicaciones básicas para el usuario. Su diseño cuenta, entre otras, con las siguientes características:

- Busca el desarrollo rápido de aplicaciones, que sean reutilizables y verdaderamente portables entre diferentes dispositivos.
- Los componentes básicos de las aplicaciones se pueden sustituir fácilmente por otros.
- Cuenta con su propia máquina virtual, Dalvik, que interpreta y ejecuta código escrito en Java.
- Permite la representación de gráficos 2D y 3D.
- Posibilita el uso de bases de datos.
- Soporta un elevado número de formatos multimedia.
- Servicio de localización GSM.
- Controla los diferentes elementos hardware: Bluetooth, Wi-Fi, cámara fotográfica o de vídeo, GPS, acelerómetro, infrarrojos, etc., siempre y cuando el dispositivo móvil lo contemple.
- Cuenta con un entorno de desarrollo muy cuidado mediante un SDK disponible de forma gratuita.
- Ofrece un *plug*-in para uno de los entornos de desarrollo más populares, Eclipse, y un emulador integrado para ejecutar las aplicaciones.

En las siguientes líneas se dará una visión global por capas de cuál es la arquitectura empleada en Android [27]. Cada una de estas capas utiliza servicios ofrecidos por las anteriores, y ofrece a su vez los suyos propios a las capas de niveles superiores.

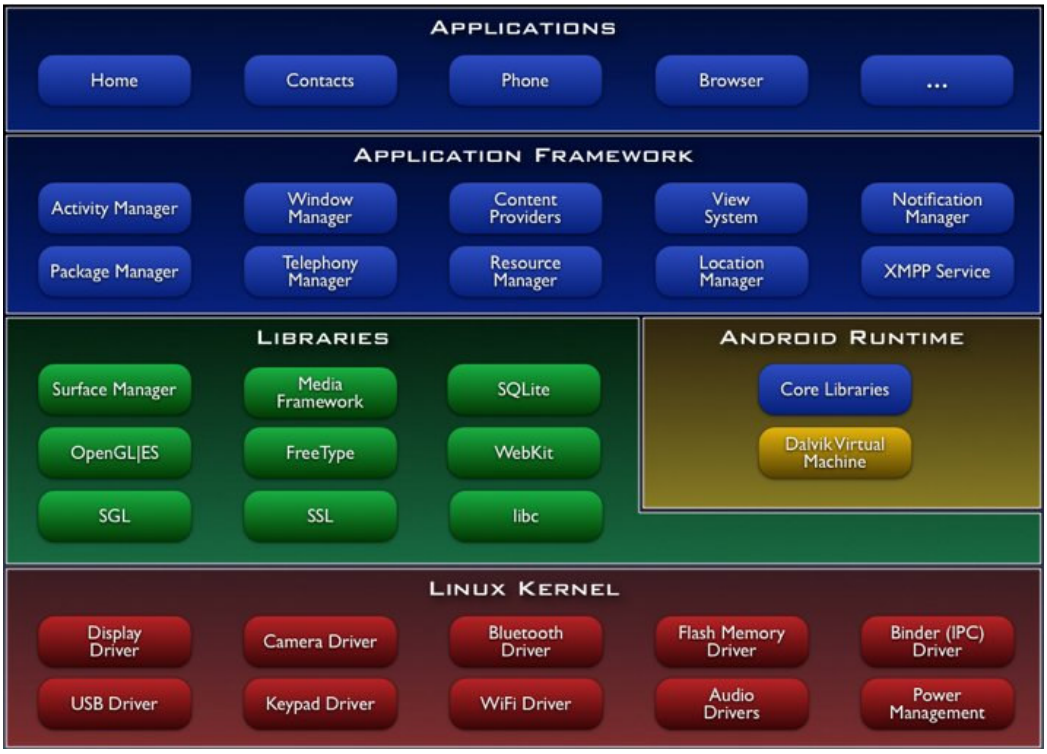


**Figura 5. Arquitectura de Android**

La capa más inmediata es la corresponde al **núcleo de Android**. Android utiliza el núcleo de Linux 2.6 como una capa de abstracción para el hardware disponible en los dispositivos móviles. Esta capa contiene los drivers necesarios para que cualquier componente hardware pueda ser utilizado mediante las llamadas correspondientes. Siempre que un fabricante incluya un nuevo elemento de hardware, lo primero que se debe realizar para que pueda ser utilizado desde Android es crear las librerías de control o drivers necesarios dentro de este kernel de Linux embebido en el propio Android.

La elección de Linux 2.6 se ha debido principalmente a dos razones: la primera, su naturaleza de código abierto y libre se ajusta al tipo de distribución que se buscaba para Android (cualquier otra opción comercial disponible hoy día hubiera comprometido la licencia de Apache); la segunda es que este kernel de Linux incluye de por sí numerosos drivers, además de contemplar la gestión de memoria, gestión de procesos, módulos de seguridad, comunicación en red y otras muchas responsabilidades propias de un sistemas operativo.

La siguiente capa se corresponde con las **librerías** utilizadas por Android. Éstas han sido escritas utilizando C/C++ y proporcionan a Android la mayor parte de sus capacidades más características. Junto al núcleo basado en Linux, estas librerías constituyen el corazón de Android.

Entre las librerías más importantes de este nivel, se pueden mencionar las siguientes:

- La librería *libc* incluye todas las cabeceras y funciones según el estándar del lenguaje C. Todas las demás librerías se definen en este lenguaje.
- La librería *Surface Manager* es la encargada de componer los diferentes elementos de navegación de pantalla. Gestiona también las ventanas pertenecientes a las distintas aplicaciones activas en cada momento.
- *OpenGL/SL* y *SGL* representan las librerías gráficas y, por tanto, sustentan la capacidad gráfica de Android. *OpenGL/SL* maneja gráficos en 3D y permite utilizar, en caso de que esté disponible en el propio dispositivo móvil, el hardware encargado de proporcionar gráficos 3D. Por otro lado, *SGL* proporciona gráficos en 2D, por lo que será la librería más habitualmente utilizada por la mayoría de las aplicaciones. Una característica importante de la capacidad gráfica de Android es que es posible desarrollar aplicaciones que combinen gráficos en 3D y 2D.
- La librería *Media Libraries* proporciona todos los códecs necesarios para el contenido multimedia soportado en Android (vídeo, audio, imágenes estáticas y animadas, etc.)
- *FreeType*, permite trabajar de forma rápida y sencilla con distintos tipos de fuentes.
- La librería *SSL* posibilita la utilización de dicho protocolo para establecer comunicaciones seguras.
- A través de la librería *SQLite*, Android ofrece la creación y gestión de bases de datos relacionales, pudiendo transformar estructuras de datos en objetos fáciles de manejar por las aplicaciones.
- La librería *WebKit* proporciona un motor para las aplicaciones de tipo navegador, y forma el núcleo del actual navegador incluido por defecto en la plataforma Android.

Al mismo nivel que las librerías de Android se sitúa el **entorno de ejecución**. Éste lo constituyen las *Core Libraries*, que son librerías con multitud de clases de Java, y la máquina virtual Dalvik.

Los dos últimos niveles de la arquitectura de Android están escritos enteramente en Java. El ***framework* de aplicaciones** representa fundamentalmente el conjunto de herramientas de desarrollo de cualquier aplicación. Toda aplicación que se desarrolle para Android, ya sean las propias del dispositivo, las desarrolladas por Google o terceras compañías, o incluso las que el propio usuario cree, utilizan el mismo conjunto de API y el mismo *framework*, representado por este nivel.

Entre las API más importantes ubicadas aquí, se pueden encontrar las siguientes:

- *Activity Manager*, importante conjunto de API que gestiona el ciclo de vida de las aplicaciones en Android (del que se hablará más adelante).
- *Window Manager*, gestiona las ventanas de las aplicaciones y utiliza la librería ya vista *Surface Manager*.

- *Telephone Manager*, incluye todas las API vinculadas a las funcionalidades propias del teléfono (llamadas, mensajes, etc.)
- *Content Providers*, permite a cualquier aplicación compartir sus datos con las demás aplicaciones de Android. Por ejemplo, gracias a esta API la información de contactos, agenda, mensajes, etc. será accesible para otras aplicaciones.
- *View System*, proporciona un gran número de elementos para poder construir interfaces de usuario (GUI), como listas, mosaicos, botones, *check-boxes*, tamaño de ventanas, control de las interfaces mediante tacto o teclado, etc. Incluye también algunas vistas estándar para las funcionalidades más frecuentes.
- *Location Manager*, posibilita a las aplicaciones la obtención de información de localización y posicionamiento, y funcionar según ésta.
- *Notification Manager*, mediante el cual las aplicaciones, usando un mismo formato, comunican al usuario eventos que ocurran durante su ejecución: una llamada entrante, un mensaje recibido, conexión Wi-Fi disponible, ubicación en un punto determinado, etc. Si llevan asociada alguna acción, en Android denominada *Intent*, (por ejemplo, atender una llamada recibida) ésta se activa mediante un simple clic.
- *XMPP Service*, colección de API para utilizar este protocolo de intercambio de mensajes basado en XML.

El último nivel del diseño arquitectónico de Android son las **aplicaciones**. Éste nivel incluye tanto las incluidas por defecto de Android como aquellas que el usuario vaya añadiendo posteriormente, ya sean de terceras empresas o de su propio desarrollo. Todas estas aplicaciones utilizan los servicios, las API y librerías de los niveles anteriores.

## 3.2 La máquina virtual Dalvik

En Android, todas las aplicaciones se programan en el lenguaje Java y se ejecutan mediante una máquina virtual de nombre Dalvik, específicamente diseñada para Android. Esta máquina virtual ha sido optimizada y adaptada a las peculiaridades propias de los dispositivos móviles (menor capacidad de proceso, baja memoria, alimentación por batería, etc.) y trabaja con ficheros de extensión .dex (*Dalvik Executables*). Dalvik no trabaja directamente con el *bytecode* de Java, sino que lo transforma en un código más eficiente que el original, pensado para procesadores pequeños.

Gracias a la herramienta "dx", esta transformación es posible: los ficheros .class de Java se compilan en ficheros .dex, de forma que cada fichero .dex puede contener varias clases. Después, este resultado se comprime en un único archivo de extensión .apk (*Android Package*), que es el que se distribuirá en el dispositivo móvil. Dalvik permite varias instancias simultáneas de la máquina virtual, y a diferencia de otras máquinas virtuales, está basada en registros y no en pila, lo que implica que las instrucciones son más reducidas y el número de accesos a memoria es menor [28]. Así mismo, Dalvik no permite la compilación *Just-in-Time*.

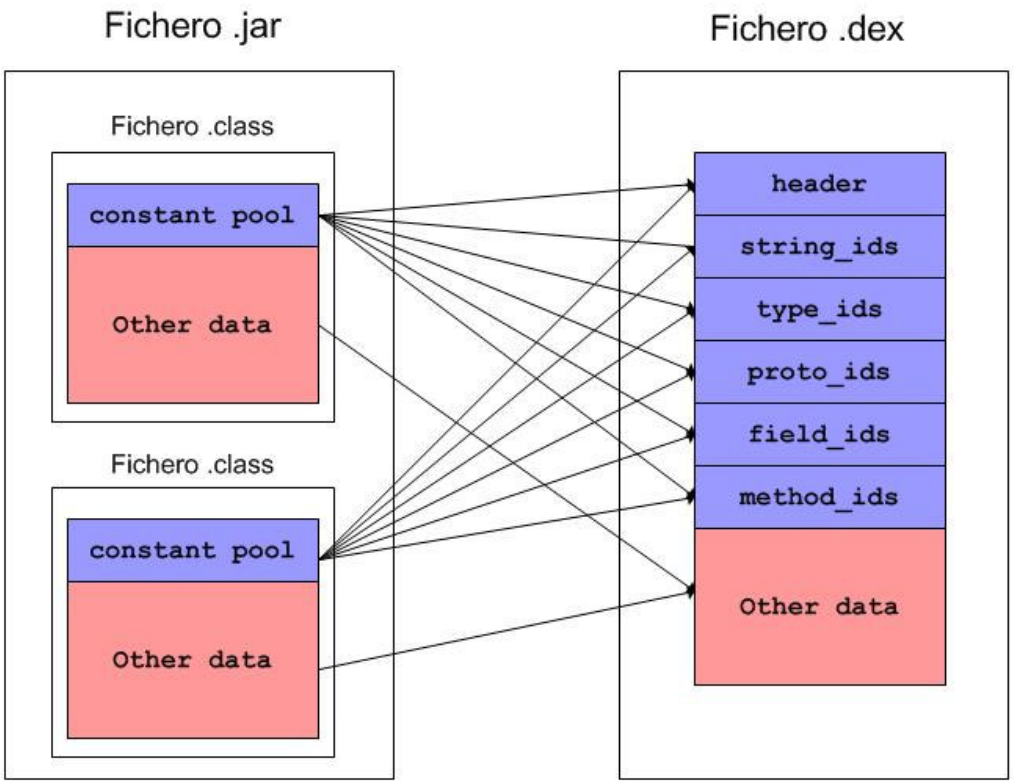


**Figura 6. Formato de un fichero .dex**

Según los responsables del proyecto, la utilización de esta máquina virtual responde a un deseo de mejorar y optimizar la ejecución de aplicaciones en dispositivos móviles, así como evitar la fragmentación de otras plataformas como Java ME. A pesar de esta aclaración oficial, no ha pasado inadvertido que con esta maniobra Android ha esquivado tener que utilizar directamente Java ME, evitando así que los futuros desarrolladores de aplicaciones tengan que comprar ninguna licencia, ni tampoco que publicar el código fuente de sus productos [29]. El lenguaje Java utilizado en Android no sigue ningún JSR determinado, y Google se afana en recordar que Android no es tecnología Java, sino que simplemente utiliza este lenguaje en sus aplicaciones [30].

## *3.3 Componentes de una aplicación*

Todas las aplicaciones en Android pueden **descomponerse en** cuatro tipos de **bloques o componentes principales**. Cada aplicación será una combinación de uno o más de estos componentes, que deberán ser declarados de forma explícita en un fichero con formato XML denominado "AndroidManifest.xml", junto a otros datos asociados como valores globales, clases que implementa, datos que puede manejar, permisos, etc. Este fichero

es básico en cualquier aplicación en Android y permite al sistema desplegar y ejecutar correctamente la aplicación.

A continuación se exponen los cuatro tipos de componentes en los que puede dividirse una aplicación para Android.

### 3.3.1 Activity

Sin duda es el componente más habitual de las aplicaciones para Android. Un componente *Activity* refleja una determinada actividad llevada a cabo por una aplicación, y que lleva asociada típicamente una ventana o interfaz de usuario; es importante señalar que no contempla únicamente el aspecto gráfico, sino que éste forma parte del componente *Activity* a través de vistas representadas por clases como `View` y sus derivadas. Este componente se implementa mediante la clase de mismo nombre `Activity`.

La mayoría de las aplicaciones permiten la ejecución de varias acciones a través de la existencia de una o más pantallas. Por ejemplo, piénsese en una aplicación de mensajes de texto. En ella, la lista de contactos se muestra en una ventana. Mediante el despliegue de una segunda ventana, el usuario puede escribir el mensaje al contacto elegido, y en otra tercera puede repasar su historial de mensajes enviados o recibidos. Cada una de estas ventanas deberías estar representada a través de un componente *Activity*, de forma que navegar de una ventana a otra implica lanzar una actividad o dormir otra. Android permite controlar por completo el ciclo de vida de los componentes *Activity*.

Muy vinculado a este componente se encuentran los *Intents*, una interesante novedad introducida por Android. Un ***Intent*** consiste básicamente en la voluntad de realizar alguna acción, generalmente asociada a unos datos. Lanzando un *Intent*, una aplicación puede **delegar el trabajo** en otra, de forma que el sistema se encarga de buscar qué aplicación entre las instaladas es la que puede llevar a cabo la acción solicitada. Por ejemplo, abrir una URL en algún navegador web, o escribir un correo electrónico desde algún cliente de correo.

### 3.3.2 Broadcast Intent Receiver

Un componente *Broadcast Intent Receiver* se utiliza para lanzar alguna ejecución dentro de la aplicación actual cuando un determinado evento se produzca (generalmente, abrir un componente *Activity*). Por ejemplo, una llamada entrante o un SMS recibido. No tiene interfaz de usuario asociada, pero puede utilizar el API *Notification Manager*, mencionada anteriormente, para avisar al usuario del evento producido a través de la barra de estado del dispositivo móvil. Este componente se implementa a través de una clase de nombre `BroadcastReceiver`.

Para que *Broadcast Intent Receiver* funcione, no es necesario que la aplicación en cuestión sea la aplicación activa en el momento de producirse el evento. El sistema lanzará la aplicación si es necesario cuando el evento monitorizado tenga lugar.

### 3.3.3 Service

Un componente *Service* representa una aplicación ejecutada sin interfaz de usuario, y que generalmente tiene lugar en segundo plano mientras otras aplicaciones (éstas con interfaz) son las que están activas en la pantalla del dispositivo.

Un ejemplo típico de este componente es un reproductor de música. La interfaz del reproductor muestra al usuario las distintas canciones disponibles, así como los típicos botones de reproducción, pausa, volumen, etc. En el momento en el que el usuario reproduce una canción, ésta se escucha mientras se siguen visionando todas las acciones anteriores, e incluso puede ejecutar una aplicación distinta sin que la música deje de sonar. La interfaz de usuario del reproductor sería un componente *Activity*, pero la música en reproducción sería un componente *Service*, porque se ejecuta en *background*. Este elemento está implementado por la clase de mismo nombre `Service`.

### 3.3.4 Content Provider

Con el componente *Content Provider*, cualquier aplicación en Android puede almacenar datos en un fichero, en una base de datos SQLite o en cualquier otro formato que considere. Además, estos datos pueden ser compartidos entre distintas aplicaciones. Una clase que implemente el componente *Content Provider* contendrá una serie de métodos que permite almacenar, recuperar, actualizar y compartir los datos de una aplicación.

Existe una colección de clases para distintos tipos de gestión de datos en el paquete `android.provider`. Además, cualquier formato adicional que se quiera implementar deberá pertenecer a este paquete y seguir sus estándares de funcionamiento.

## *3.4 Ciclo de vida de las aplicaciones Android*

En Android, cada aplicación se ejecuta en su propio proceso. Esto aporta beneficios en cuestiones básicas como seguridad, gestión de memoria, o la ocupación de la CPU del dispositivo móvil. Android se ocupa de lanzar y parar todos estos procesos, gestionar su ejecución y decidir qué hacer en función de los recursos disponibles y de las órdenes dadas por el usuario.

El usuario desconoce este comportamiento de Android. Simplemente es consciente de que mediante un simple clic pasa de una a otra aplicación y puede volver a cualquiera de ellas en el momento que lo desee. No debe preocuparse sobre cuál es la aplicación que realmente está activa, cuánta memoria está consumiendo, ni si existen o no recursos

suficientes para abrir una aplicación adicional. Todo eso son tareas propias del sistema operativo.

Android lanza tantos procesos como permitan los recursos del dispositivo. Cada proceso, correspondiente a una aplicación, estará formado por una o varias actividades independientes (componentes *Activity*) de esa aplicación. Cuando el usuario navega de una actividad a otra, o abre una nueva aplicación, el sistema duerme dicho proceso y realiza una copia de su estado para poder recuperarlo más tarde. El proceso y la actividad siguen existiendo en el sistema, pero están dormidos y su estado ha sido guardado. Es entonces cuando crea, o despierta si ya existe, el proceso para la aplicación que debe ser lanzada, asumiendo que existan recursos para ello.

Cada uno de los componentes básicos de Android tiene un ciclo de vida bien definido; esto implica que el desarrollador puede controlar en cada momento en qué estado se encuentra dicho componente, pudiendo así programar las acciones que mejor convengan. El componente *Activity*, probablemente el más importante, tiene un ciclo de vida como el mostrado en la Figura 3.

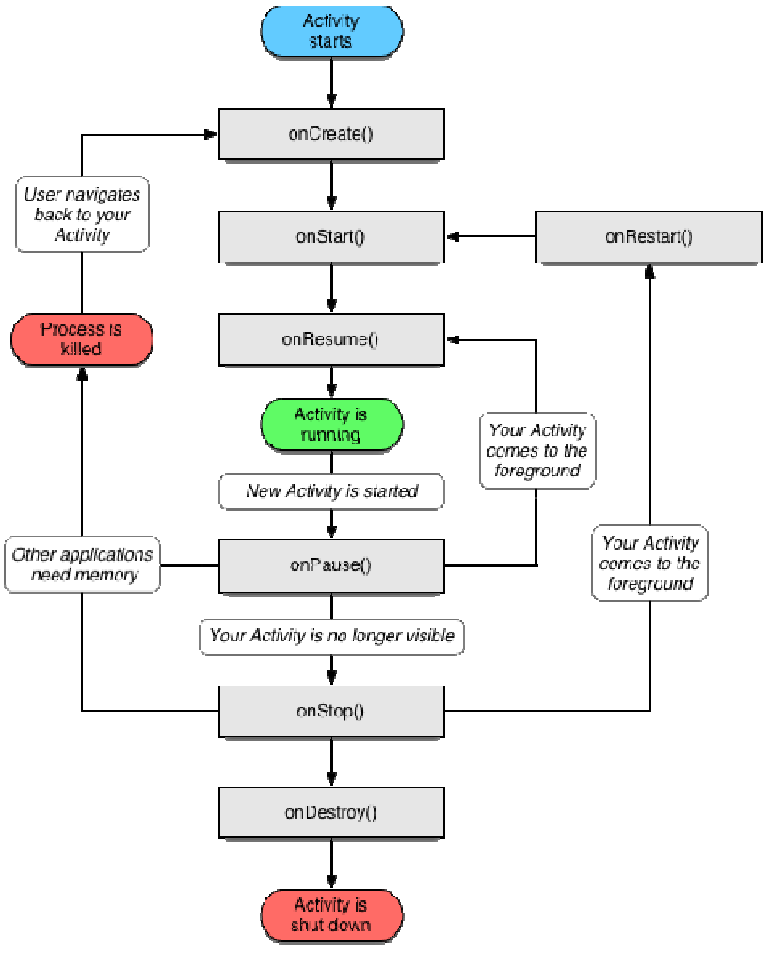


**Figura 7. Ciclo de vida de un objeto *Activity***

De la Figura 3 pueden sacarse las siguientes conclusiones:

- **`onCreate()`**, **`onDestroy()`**: abarcan todo el ciclo de vida. Cada uno de estos métodos representan el principio y el fin de la actividad.

- **`onStart()`**, **`onStop()`**: representan la parte visible del ciclo de vida. Desde `onStart()` hasta `onStop()`, la actividad será visible para el usuario, aunque es posible que no tenga el foco de acción por existir otras actividades superpuestas con las que el usuario está interactuando. Pueden ser llamados múltiples veces.

- **`onResume()`**, **`onPause()`**: delimitan la parte útil del ciclo de vida. Desde `onResume()` hasta `onPause()`, la actividad no sólo es visible, sino que además tiene el foco de la acción y el usuario puede interactuar con ella.

Tal y como se ve en el diagrama de la Figura 7, el proceso que mantiene a esta *Activity* puede ser eliminado cuando se encuentra en `onPause()` o en `onStop()`, es decir, cuando no tiene el foco de la aplicación. Android nunca elimina procesos con los que el usuario está interactuando en ese momento. Una vez se elimina el proceso, el usuario desconoce dicha situación y puede incluso volver atrás y querer usarlo de nuevo. Entonces el proceso se restaura gracias a una copia y vuelve a estar activo como si no hubiera sido eliminado. Además, la *Activity* puede haber estado en segundo plano, invisible, y entonces es despertada pasando por el estado `onRestart()`.

Pero, ¿qué ocurre en realidad cuando no existen recursos suficientes? Obviamente, los recursos son siempre limitados, más aun cuando se está hablando de dispositivos móviles. En el momento en el que Android detecta que no hay los recursos necesarios para poder lanzar una nueva aplicación, analiza los procesos existentes en ese momento y elimina los procesos que sean menos prioritarios para poder liberar sus recursos.

Cuando el usuario regresa a una actividad que está dormida, el sistema simplemente la despierta. En este caso, no es necesario recuperar el estado guardado porque el proceso todavía existe y mantiene el mismo estado. Sin embargo, cuando el usuario quiere regresar a una aplicación cuyo proceso ya no existe porque se necesitaba liberar sus recursos, Android lo crea de nuevo y utiliza el estado previamente guardado para poder restaurar una copia fresca del mismo. Como se ya ha explicado, el usuario no percibe esta situación ni conoce si el proceso ha sido eliminado o está dormido.

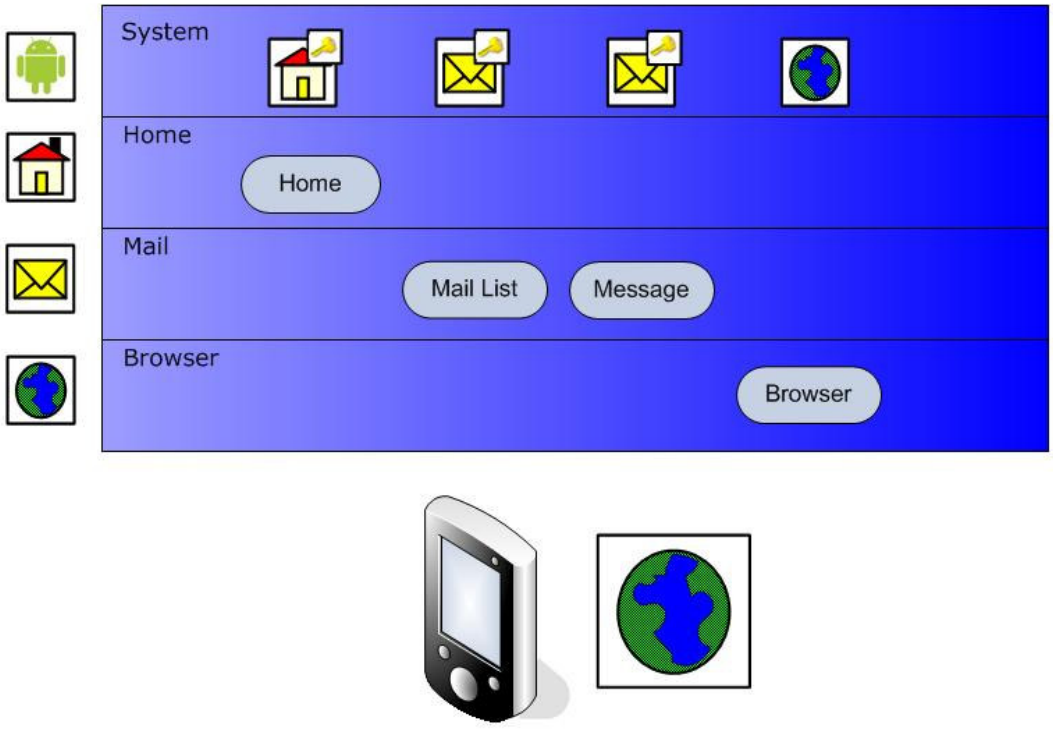


**Figura 8. Ciclo de vida de una aplicación: ejemplo (I)**

Sirva como ejemplo el caso mostrado en la **Figura 1**. En ella se representa el estado del sistema tras la ejecución de varias aplicaciones [31]. El camino que ha seguido el hipotético usuario ha sido el siguiente:

1. Abrir el menú principal de Android.
2. Seleccionar un cliente de correo, que se abre y muestra la bandeja de entrada.
3. Abrir uno de los nuevos mensaje recibidos.
4. Pinchar dentro del cuerpo del mensaje una URL.
5. El navegador web se abre mostrando dicha dirección.

En dicha figura existen 4 procesos en el sistema, representados mediante barras horizontales apiladas. Además, la representación de un dispositivo móvil con un icono al lado muestra cuál sería la aplicación activa en ese momento para el usuario:

- El primero de los procesos, *System*, representa el estado del sistema; a través del API *Activity Manager*, este proceso gestiona el ciclo de vida de los demás procesos. El usuario ha ido abriendo aplicaciones y lanzando actividades, y en este instante vemos que *System* monitoriza las siguientes actividades:

    a) el menú principal de Android.
    b) la bandeja de entrada del cliente de correo.
    c) el mensaje que el usuario ha abierto.

d) el navegador web, que muestra el hiperenlace accedido.

La actividad actualmente visible y ejecuta es el navegador web. Las demás actividades están dormidas y su estado ha sido guardado (nótese en la Figura 8 el icono con un llave, adjunto a los iconos de estas actividades del proceso *System*). De momento, no ha sido necesario eliminar ningún proceso.

- El proceso *Home* se corresponde con el menú principal del dispositivo, y tiene una única actividad llamada *Home*. Esta simplemente muestra al usuario las aplicaciones disponibles.

- El proceso *Mail* representa al cliente de correo. Cuenta con dos actividades abiertas, *Mail List*, correspondiente a la bandeja de entrada, y *Message*, que referencia al mensaje abierto.

- El cuarto y último proceso, *Browser*, es el navegador web y tiene también una única actividad *Browser* que muestra al usuario el hiperenlace pinchado desde el mensaje de correo.

Actualmente, *Browser* es la única actividad ejecutándose y visible para el usuario. Las demás actividades están dormidas y sus respectivos procesos están parados pero existen en memoria principal. Los recursos están plenamente ocupados y no hay suficientes para poder crear ningún otro proceso, lo que significa que no se pueden abrir más aplicaciones ni lanzar actividades sin eliminar alguno de los existentes.

Supóngase que, en este estado, el usuario lee en la página mostrada por el navegador una dirección que desea poder ubicar a través de una aplicación de localización geográfica llamada *Maps*. En ese momento, Android detecta que no existen recursos y se ve obligado a evaluar la prioridad de cada uno de los procesos existentes para eliminar los necesarios según los recursos requeridos para *Maps*. En la **Figura 3** se muestra el estado del sistema tras la apertura de la nueva aplicación.

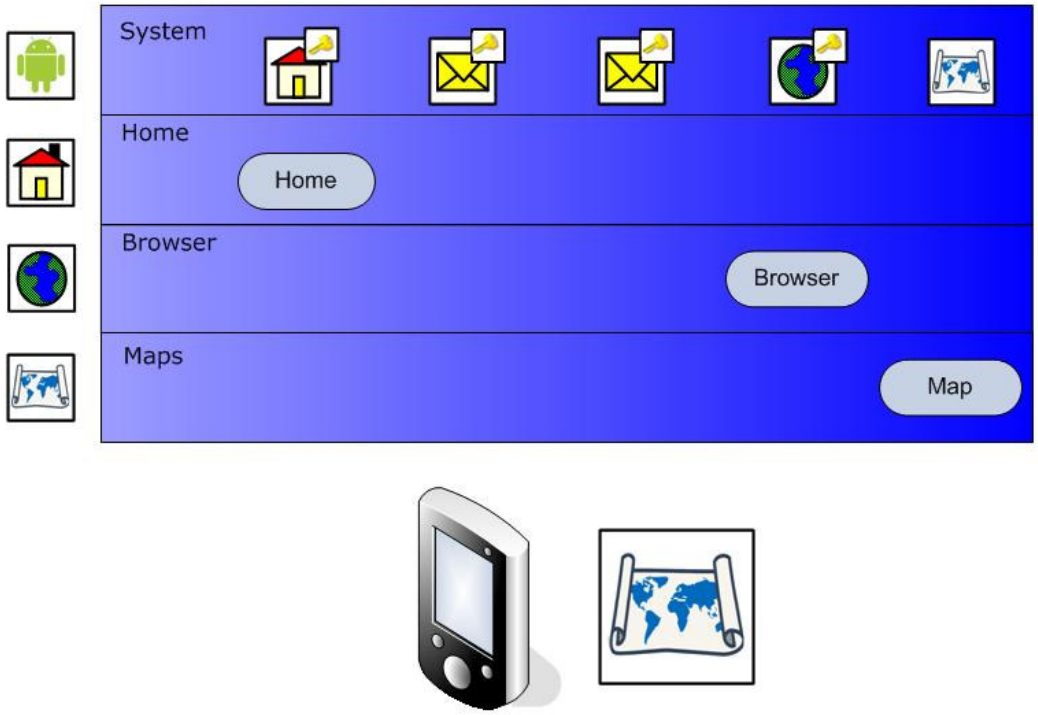


**Figura 9. Ciclo de vida de una aplicación: ejemplo (II)**

En el ejemplo, después de evaluar la situación, el sistema ha optado por eliminar el proceso correspondiente al cliente de correo, terminando así con las actividades correspondientes a *Mail List* y *Message*. Los recursos disponibles tras su eliminación permiten crear el proceso y actividad necesarios, y abrir así la aplicación *Map*. Ésta pasa a ser la aplicación activa para el usuario.

Nótese que los estados correspondientes a las actividades del cliente de correo permanecen intactos en el proceso *System*, para poder asegurar que, en caso de que el usuario decida volver a ellas, se pueda recuperar una copia reciente como si sus procesos no hubieran sido eliminados de memoria.

## 3.5 Política de eliminación de procesos

Tal y como se explica en el apartador anterior, cada aplicación Android se ejecuta en su propio proceso. Este proceso se crea cada vez que una aplicación necesita ejecutar parte de su código, y seguirá existiendo hasta que la aplicación finalice o hasta que el sistema necesite utilizar parte de sus recursos para otra aplicación considerada prioritaria.

Por ello, es importante saber cómo los componentes de una aplicación en Android (*Activity*, *Broadcast Intent Receiver*, *Service* y *Content Provider*) determinan e influyen en el ciclo de vida de la aplicación. No usar los componentes correctamente a la hora de

construir una aplicación puede significar que el sistema operativo la termine cuando en realidad está haciendo una tarea importante para el usuario.

Android construye una jerarquía donde evalúa la clase de componentes que están ejecutándose y el estado de los mismos. En orden de importancia, serían los siguientes:

1. **Procesos en primer plano**: aquellos necesarios para lo que el usuario está haciendo en ese momento. Un proceso se encuadra en esa categoría si cumple alguna de las siguientes condiciones:
    a. tiene un componente *Activity* ejecutándose, con la que el usuario está interactuando.
    b. tiene un componente *Broadcast Intent Receiver* ejecutándose.
    c. ha lanzado algún otro proceso que tiene un componente *Service* ejecutándose en el momento

    Idealmente, sólo debería haber algunos de estos procesos en el sistema, y su eliminación debería darse únicamente en casos extremos en los que la falta de recursos impide siguen ejecutándolos todos.

2. **Procesos visibles**: aquellos procesos que contienen un componente *Activity* visible en la pantalla, pero no con el foco de actividad en ese momento.

3. **Procesos de servicio**: aquellos procesos que tienen un componente *Service* y que están ejecutándose en *background.* Aunque no sean visibles directamente al usuario, desempeñan tareas sí percibidas por este.

4. **Procesos en segundo plano**: procesos con un componente *Activity*, que no son visibles al usuario. Estos procesos no tienen una importancia directa para el usuario en ese momento.

5. **Procesos vacíos**: procesos que ya no ejecutan ninguna actividad, pero que se mantienen en memoria para agilizar una posible nueva llamada por parte del usuario.

Según esta jerarquía, Android prioriza los procesos existentes en el sistema y decide cuáles han de ser eliminados, con el fin de liberar recursos y poder lanzar la aplicación requerida.

## *3.6 Seguridad en Android*

En Android cada aplicación se ejecuta en su propio proceso. La mayoría de las medidas de seguridad entre el sistema y las aplicaciones deriva de los estándares de Linux 2.6, cuyo kernel, recuérdese, constituye el núcleo principal de Android.

Cada proceso en Android constituye lo que se llama un cajón de arena o *sandbox*, que proporciona un entorno seguro de ejecución. Por defecto, ninguna aplicación tiene

permiso para realizar ninguna operación o comportamiento que pueda impactar negativamente en la ejecución de otras aplicaciones o del sistema mismo. Por ejemplo, acciones como leer o escribir ficheros privados del usuario (contactos, teléfonos, etc.), leer o escribir ficheros de otras aplicaciones, acceso de red, habilitación de algún recurso hardware del dispositivo, etc., no están permitidas. La única forma de poder saltar estas restricciones impuestas por Android, es mediante la declaración explícita de un permiso que autorice a llevar a cabo una determinada acción habitualmente prohibida.

Además, en Android es obligatorio que cada aplicación esté firmada digitalmente mediante un certificado, cuya clave privada sea la del desarrollador de dicha aplicación. No es necesario vincular a una autoridad de certificado, el único cometido del certificado es crear una relación de confianza entre las aplicaciones. Mediante la firma, la aplicación lleva adjunta su autoría.

Para establecer un permiso para una aplicación, es necesario declarar en el manifiesto uno o más elementos `<uses-permission>` donde se especifica el tipo de permiso que se desea habilitar. Por ejemplo, si se quisiera permitir que una aplicación pueda monitorizar mensajes SMS entrantes, en el fichero "AndroidManifest.xml" se encontraría algo como lo que sigue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.android.app.myapp" >

      <uses-permission
            android:name="android.permission.RECEIVE_SMS" />

</manifest>
```

**Código 1. Ejemplo de permiso para manipular mensajes SMS**

En la clase `android.Manifest.permission` se especifican todos los posibles permisos que se pueden conceder a una aplicación: utilización de Wi-Fi, Bluetooth, llamadas telefónicas, cámara, Internet, mensajes SMS y MMS, vibrador, etc.

Los permisos pueden definirse de forma más completa que la anteriormente mostrada. El elemento `<uses-permission>` contempla una serie de atributos que definen y matizan el alcance del permiso dado:

- `android:name`: especificación del permiso que se pretende conceder. Debe ser un nombre de alguno de los listados en la clase `android.Manifest.permission`.

- `android:label`: una etiqueta o nombre convencional fácilmente legible para el usuario.

- `android:permissionGroup`: permite especificar un grupo asociado al permiso. Los posibles grupos se encuentran listados en la clase `android.Manifest.permission_group` y pueden tener valores como `ACCOUNTS` (cuentas válidas de Google), `COST_MONEY` (acciones que llevan vinculadas un pago) o `PHONE_CALLS` (acciones relacionadas con llamadas), entre otros.

- `android:protectionLevel`: determina el nivel de riesgo del permiso, y en función del mismo influye en cómo el sistema otorga o no el permiso a la aplicación. Oscila entre valores desde el 0 hasta el 3.

- `android:description`: descripción textual del permiso.

- `android:icon`: icono gráfico que puede ser asociado al permiso.

## *3.7 Gestión de la información*

A diferencia de lo que ocurre en otros sistemas, en Android cualquier repositiorio de datos, como por ejemplo archivos, son privados y únicamente accesibles por la aplicación a la que pertenecen. Sin embargo, esto no implica necesariamente que no puedan ser compartidos o accedidos por otras aplicaciones; Android facilita una serie de mecanismos que posibilitan en ciertas circunstancias el acceso a dicha información.

En los siguientes apartados se explican las distintas formas que tiene una aplicación para almacenar y recuperar información, así como los mecanismos a utilizar en caso de desear compartirla con las demás aplicaciones.

### 3.7.1 Preferencias de usuario

Las preferencias son todos aquellos valores asociados a una determinada aplicación y que permiten adaptar ésta a los gustos o necesidades del usuario. Solamente se puede acceder a las preferencias dentro de un mismo paquete; Android no permite compartir estos valores con otras aplicaciones.

Para poder compartir las preferencias entre todos los componentes que forman la aplicación, dentro del mismo paquete, debe asignarse un nombre al conjunto de valores que forman las preferencias de usuario. Después, éstas se pueden recuperar a lo largo de todo el paquete haciendo una llamada a `Context.getSharedPreferences()`. Si las preferencias solamente serán accedidas desde un componente Activity y no es necesario compartirlas con los demás componentes, es posible omitir el nombre asociado a las preferencias y recuperarlas simplemente llamando a `Activity.getPreferences()`.

### 3.7.2 Ficheros

Para leer o escribir ficheros, Android facilita los métodos `Context.openFileInput()` y `Context.openFileOutput()`, respectivamente. Estos ficheros deben ser locales a la aplicación en curso, es decir, al igual que con las preferencias, un fichero "estándar" de datos no puede ser compartido con otras aplicaciones.

Si el fichero es estático y puede ser adjuntado a la aplicación en el momento que se compila, éste puede ser añadido como un recurso más. Su ubicación correcta será en la carpeta de recursos, concretamente en "\res\raw\nombrefichero.extension". Esto permitirá que acompañe al paquete completo y que pueda ser leído directamente con el método `Resources.openRawResource (R.raw.nombrefichero)`.

### 3.7.3 Bases de datos

Android incluye una librería de SQLite que permite crear bases de datos relacionales, navegar entre las tablas, ejecutar sentencias en SQL y otras funcionalidades propias del sistema SQLite. La base de datos resultante puede ser accedida desde el código de la aplicación como si de un objeto más se tratara, gracias a las clases contenidas en el paquete `android.database.sqlite`.

Cualquier base de datos será accesible desde los demás componentes de la misma aplicación, pero no fuera de ésta.

### 3.7.4 Acceso por red

Como es lógico, Android permite la comunicación entre dispositivos a través de una infraestructura de red, siempre y cuando ésta esté disponible. Los paquetes `java.net` y `android.net` ofrecen multitud de clases y posibilidades en este sentido. Más adelante se verán con mayor detenimiento sus capacidades.

### 3.7.5 Content Provider

Hasta ahora, se han mencionado algunas formas para almacenar y recuperar información de forma siempre local y sin poder compartirla con otras aplicaciones del mismo dispositivo. Android ofrece un mecanismo que posibilita el acceso a información de forma compartida: usando un Content Provider o proveedor de contenidos. Ésta es la única forma habilitada para facilitar datos más allá del propio paquete de la aplicación.

Por defecto, Android incluye una serie de componentes Content Provider que permiten publicar todo tipo de datos básicos que pueden resultar útiles entre aplicaciones: información de los contactos, fotografías, imágenes, vídeos, mensajes de texto, audios, etc. Todos estos Content Provider ya definidos y listos para utilizar se pueden encontrar

en el paquete `android.provider`. Además, Android ofrece la posibilidad de que el desarrollador pueda crear sus propios Content Provider.

Un Content Provider es un objeto de la clase `ContentProvider`, ubicada en el paquete `android.content`, y que almacena datos de un determinado tipo que pueden ser accedido desde cualquier aplicación. Cada elemento Content Provider tiene asociado una URI única que lo representa y a través de la cual los otros componentes de una aplicación pueden acceder a él.

Por ejemplo, la cadena "content://contacts/people/" es una URI válida que da acceso a la información de contactos del dispositivo; la cadena "content://media/external/images" es la URI identificativa de otro Content Provider que da acceso a las imágenes de un dispositivo de almacenamiento externo (tarjeta SD, por ejemplo).

Conocida la URI, cualquier componente puede acceder al correspondiente Content Provider. En el siguiente ejemplo, mostrado en el Código 2, se accede a la información de contactos del dispositivo utilizando la URI que identifica el Content Provider de los contactos. La información que se quiere obtener es el ID del contacto, su nombre y su teléfono. Para hacer una consulta a través de una URI, existen varios métodos disponibles: uno de ellos es `managedQuery()`, de la clase `Activity`. Este método requiere como parámetros la URI a la que se consulta, los campos a seleccionar, y otros valores como las cláusulas WHERE u ORDER BY, como se de una consulta SQL se tratara. El resultado consiste en un conjunto de filas que se puede recorrer a través del objeto `Cursor`, clase ubicada en el paquete `android.database`.

```
// Columnas a consultar
String[] projection = new String[] {
      Contacts.People.NAME,
      Contacts.People.NUMBER,
      Contacts.People._ID
};

// Establecer URI para acceder a los contactos
Uri contacts = "content://contacts/people/";

// Lanzar consulta
Cursor cursor = managedQuery( contacts,
      projection,
      null,
      null,
      Contacts.People.NAME + " ASC");
```

**Código 2. Ejemplo de consulta a un Content Provider**

Los mecanismos de actualización e inserción de datos son muy similares a la consulta anteriormente vista. Sin embargo, el mayor potencial de los Content Provider estriba en que el desarrollador cree los suyos propios adaptados a las necesidades de su aplicación.

Para crear un Content Provider, se deben seguir los siguientes pasos:

1. Crear una clase que extienda a `ContentProvider`.

2. Definir una constante denominada `CONTENT_URI` donde quede recogida la URI que identificará a este nuevo Content Provider. Esta cadena **siempre** ha de comenzar por el prefijo "content://" seguida de la jerarquía de nombres que se desee establecer. Por ejemplo "content://miaplicacion.misdatos"

3. Establecer el sistema de almacenamiento deseado. Lo habitual es establecer una base de datos en SQLite, pero se puede utilizar cualquier mecanismo de almacenamiento.

4. Definir también como constantes el nombre de las columnas de datos que pueden ser recuperadas, de forma que el futuro usuario de este Content Provider puede conocerlas.

5. Implementar los métodos básicos de manipulación de datos. Estos son:
   - `query()`, debe devolver un objeto `Cursor`.
   - `insert()`
   - `update()`
   - `delete()`
   - `getType()`

6. En el fichero "AndroidManifest.xml" de la aplicación debe añadirse, dentro del elemento `<application>`, la siguiente etiqueta:

```
<application>
        <provider android:name="nombre_del_content_provider"
                android:authorities="uri_asignada"
         />
</application>
```

**Código 3. Declaración en el manifiesto de un componente Content Provider**

## 3.8 API Demos

Como parte del SDK de Android, Google ha incluido un gran número de clases dentro de un bloque denominado "APIDemos". Estas clases ejemplifican de forma práctica

muchas de las características y posibilidades ofrecidas por el sistema Android. Algunas de ellas son las siguientes:

- APIDemos\app: incluye ejemplos de cómo trabajar con los principales componentes de Android, como por ejemplo *Activity* o *Service*, y algunas aplicaciones sencillas de ejemplo. En estas clases se pueden encontrar ilustraciones sobre cómo guardar el estado de una actividad cuando es dormida, usar las preferencias de usuario, devolver un valor desde otra actividad, lanzar componentes *Service* en el mismo o distinto proceso, o sobre cómo lanzar alarmas y notificaciones al usuario.

- APIDemos\View: en esta útil carpeta se pueden ver ejemplos de distintos tipos de vistas y diseños para las interfaces de usuario. Aquí se encuentran clases que ilustran la creación de formularios, distribuciones verticales y horizontales de elementos, vistas que se adaptan al contenido, ventanas con *scroll*, listados de elementos, botones compuestos por imágenes, galerías de imágenes o barras de progreso.

- APIDemos\Graphics: aquí se ofrecen ejemplos de la capacidad gráfica de Android, tanto de la definición de imágenes con movimientos como de la construcción de figuras geométricas básicas.

- APIDemos\Media: ejemplos sobre el manejo formatos multimedia, como vídeo, sonidos o imágenes.

Estos códigos de ejemplo muy recomendables cuando ya se han conocido tanto el funcionamiento más básico de Android como sus características principales, y resultan de gran utilidad para comenzar a hacerse una idea general del tipo de aplicaciones que este sistema puede soportar.

## 3.9 Funciones del emulador

El SDK de Android incluye un completo emulador que permite probar y depurar eficientemente las aplicaciones que se van desarrollando. Este emulador incluye diferentes *skins* para representar gráficamente un dispositivo móvil real, con pantalla, teclado y botones de función, así como aplicaciones de usuario preinstaladas que incluyen navegador web o visor de mapa, y una consola que permite interactuar con él a través de Telnet.

El emulador, como se ha comentado, es muy completo y escapa al propósito de este apartado su explicación detallada. Aún así, se mencionan a continuación algunas de las principales características de esta útil herramienta para el desarrollo en Android.

Puede ser lanzado tanto a través de una ventana de comandos, con el ejecutable de nombre "emulator.exe" de la carpeta "\tools" o, la que probablemente será la mejor, a través del *plug-in* para Eclipse. En este último caso, cada vez que se ejecute una

aplicación, se hará automáticamente a través del emulador incluido en el SDK. Las opciones de lanzamiento desde Eclipse se pueden revisar en *Debug/Run -> Target*.

Este emulador simula un procesador del tipo ARM y permite tener varias instancias corriendo al mismo tiempo, cada uno con una emulación distinta. Permite simular varios eventos como llamadas entrantes o mensajes SMS, incluso conectar varios emuladores entre sí (para ello es necesario redireccionar puertos, como se verá más adelante).


**Figura 10. Ejemplo de *skin* para el emulador de Android**

Una vez lanzado el emulador, puede interactuarse con éste a través de su interfaz gráfica pulsando en la pantalla del dispositivo, sobre sus botones de función o en el teclado virtual. También pueden lanzarse órdenes conectándose por Telnet al puerto donde esté escuchando, siendo el 5554 para la primera de las instancias que se lancen.

### 3.9.1 Uso de imágenes

En el emulador de Android pueden utilizarse imágenes de datos que se guardan en el propio equipo de desarrollo. Estas imágenes simulan memorias internas, tipo flash, y otras.

En concreto, el emulador dispone de tres tipos de ficheros imagen:

- **Imagen de sistema**: contiene datos y especificaciones para la ejecución, fundamentales para el emulador. Es únicamente de lectura ya que se lee al principio y no se vuelve a acceder a ella hasta la próxima ejecución. Se almacenan en la carpeta "\lib" y se llaman "kernel.img", "system.img", "ramdisk.img" y "userdata.img".

- **Imagen de ejecución**: el emulador dispone de dos ficheros imagen para lectura y escritura de datos. El primer de ellos, "userdata-qemu.img", almacena los datos del usuario como preferencias, ficheros, contenidos de la base de datos SQLite, etc. Al arrancar, el emulador comprueba si existe este fichero: si no, lo crea como una copia exacta de la imagen de sistema "userdata.img"; si existe, lo abre sin más. Si al terminar los datos han sido modificados, serán guardados en esta imagen. Si se utilizan varios emuladores, deberán ser configurados para que cada uno cuente con su propia imagen "userdata-qemu.img". La otra imagen de ejecución se denomina "sdcard.img", y se utiliza para simular dispositivos de almacenamiento extraíbles. Para crear una imagen de este tipo, que es opcional, es necesario utilizar la herramienta "mksdcard" de la carpeta "\tools".

- **Imagen temporal**: al ejecutarse el emulador, siempre se crean dos ficheros imagen para datos temporales: uno que hace una copia del sistema completo, y otro que se utiliza como caché para el navegador web y otros servicios. Ambas imágenes se eliminan al terminar la emulación.

### 3.9.2 Aspectos de red

Android incorpora a su emulador un pequeño router/firewall virtual, de forma que el sistema emulado desconoce tanto la máquina de desarrollo como las demás instancias del mismo emulador. Sin embargo, este comportamiento es configurable.

Todas las instancias del emulador tienen asignada una dirección de red de la forma 10.0.2/24. Por defecto, las direcciones definidas para cada instancia son las mostradas en la siguiente tabla:

| IP | Descripción |
|---|---|
| 10.0.2.1 | *Gateway* virtual |
| 10.0.2.2 | Máquina de desarrollo. |
| 10.0.2.3 | Servidor DNS principal. |
| 10.0.2.4 / 10.0.2.5 / 10.0.2.6 | Servidores DNS secundarios. |
| 10.0.2.15 | El sistema emulado, esto es, localhost. |

**Tabla 2. Direcciones de red en el emulador**

Para poder interactuar libremente con la aplicación emulada, es necesario redireccionar los puertos correspondientes a fin de saltar las restricciones impuestas por el router virtual del emulador. Para ello se utiliza el comando `redir`, después de haberse conectado por Telnet. Véase siguiente ejemplo, mostrado en el Código 4:

```
> telnet localhost 5554
Android console: type 'help' for a list of commands
OK
redir add tcp:5000:6000
OK
```

**Código 4. Ejemplo de redirección de puertos en el emulador**

En este ejemplo en primer lugar se conecta al emulador por Telnet, usando el puerto 5554, que es el utilizado para la primera de las instancias del emulador. A continuación se invoca el comando `redir`, y se añade una nueva regla mediante `add`. Dicha regla especifica que todas las conexiones TCP llegadas al puerto 5000 de la máquina de desarrollo se deriven al puerto 6000 del emulador. También puede utilizarse con el protocolo UDP.

Esta sencilla regla puede utilizarse para poder comunicar dos instancias del emulador. Supóngase el siguiente escenario:

1. A es la máquina de desarrollo.
2. B es una instancia del emulador Android, ejecutándose en A. En B existe una aplicación que hace de servidor y escucha por el puerto 80.
3. C es otra instancia del emulador, también ejecutándose en A. En C existe una aplicación que hace de cliente y desea conectarse con el servidor en B.

Para poder cumplir estos supuestos, se debe establecer esta configuración:

1. B debe escuchar en la dirección 10.2.0.15:80, es decir, en su localhost.
2. B debe tener una regla del tipo `redir add tcp 8080:80`
3. C debe conectarse a la dirección 10.0.2.2:8080.

De esta forma, C siempre se conectará a la máquina de desarrollo en su puerto 8080, y la regla puesta en B hará que esta conexión TCP se derive a ella misma por el puerto 80. Así, ambos emuladores pueden establecer comunicación.

### 3.9.3 Órdenes desde consola

Aunque el emulador ya esté funcionando, hay muchos aspectos de este que pueden ser configurados al momento utilizando la consola a través de un Telnet al puerto 5554 y posteriores, según la instancia del emulador deseada.

Algunos de los comandos aceptados por la consola son los siguientes:

- `help`: muestra una lista de los comandos disponibles.
- `event`: permite enviar diferentes tipos de eventos al emulador.
- `geo`: permite establecer una localización fija mediante coordenadas. Será la que devuelva el dispositivo de GPS al ser invocado desde una aplicación.
- `gsm`: emula llamadas telefónicas.
- `kill`: finaliza la instancia del emulador.
- `network`: diferentes aspectos relacionados con el control de los distintos tipos de conexiones (GSM, GPRS, UMTS, EDGE), como los valores de latencia.
- `power`: controla la simulación del nivel de batería del dispositivo móvil.
- `quit`/`exit`: cierra la conexión con el emulador y la consola.
- `redir`: permite redireccionar puertos entre el emulador y la máquina de desarrollo.
- `sms`: permite enviar mensajes SMS al emulador.
- `vm`: control de la máquina virtual Dalvik.
- `window`: configuración del *skin* del emulador.

### 3.9.4 Dalvik Debug Monitor

El emulador incluye una potente herramienta llamada Dalvik Debug Monitor, que será de gran ayuda para testear las aplicaciones. Esta aplicación permite hacer un seguimiento de los puertos utilizados, ver mensajes de log, información acerca de los procesos en ejecución, gestión de memoria, eventos entrantes como llamadas y SMS, así como otras muchísimas opciones orientados a la depuración.

Para utilizarla, debe ser lanzado en primer lugar un emulador. Entonces, en la carpeta "Tools" se debe ejecutar un fichero de nombre "ddms.exe". La ventana que se abre en ese momento es similar a la mostrada a continuación:

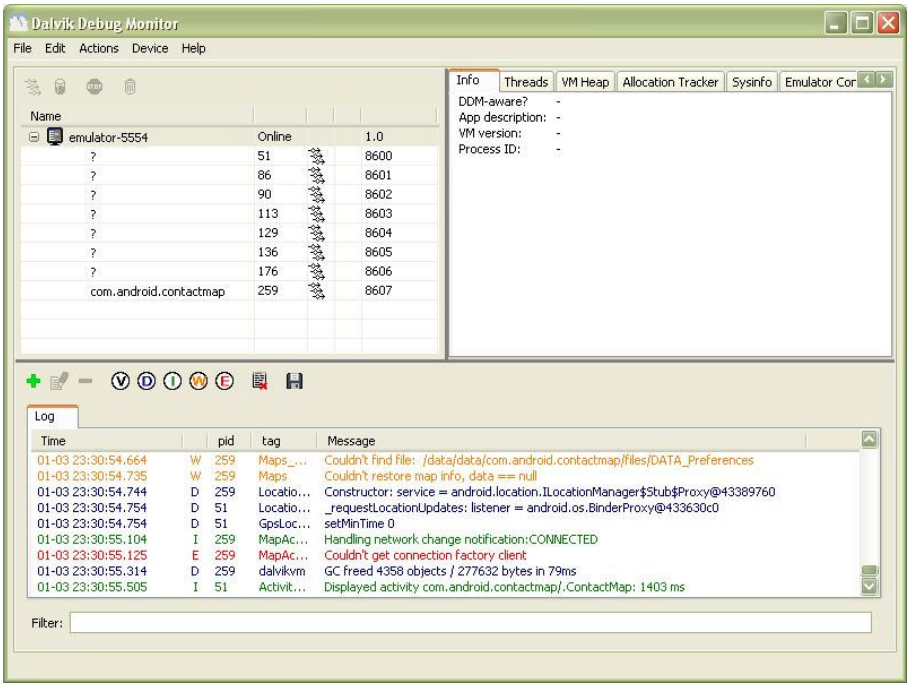


**Figura 11. Dalvik Debug Monitor**

Una de las opciones ofrecidas más útiles, y que probablemente se quieran utilizar durante la depuración de las distintas aplicaciones, es la posibilidad de imprimir mensajes de traza. Para tal efecto está destinada la pestaña inferior de las mostradas en la Figura 11. En cada ejecución existe siempre una pestaña de nombre "Log" que se utiliza para indicar al desarrollador los diferentes procesos y eventos que tienen lugar durante la ejecución, utilizando un código de colores para diferenciar su naturaleza.

Una de los paquetes de Android, `android.util`, contiene una clase llamada `Log` que permite imprimir mensajes de debug en la pestaña antes referenciada. El método `Log.d(title,message)` permite escribir estos mensajes, que después Dalvik Debug Monitor puede capturar. En dicho método, `title` representa el título o etiqueta que se desee dar, a fin de poder filtrar después todos los mensajes de dicha categoría; por otro lado, `message` indica el mensaje concreto que se quiere imprimir bajo esta etiqueta.

## 3.10 Instalación de Eclipse con el SDK de Android

En este apartado se exponen los pasos necesarios para empezar a desarrollar y a entender aplicaciones para Android. Las instrucciones de instalación aquí descritas se basan en el sistema operativo Windows XP o Windows Vista, y en el entorno de desarrollo Eclipse Classic versión 3.4, conocida como Ganymede. Aunque esta guía de instalación no los contempla, el SDK de Android también puede correr en otros sistemas operativos como Mac OS X o Linux (verificado para Ubuntu 6.06 LTS).

### 3.10.1 Descargar el SDK de Android

Android es una plataforma de software libre, por lo que cuenta con un SDK disponible para todo desarrollador que lo desee que incluye, entre otros elementos, el conjunto completo de API que este sistema soporta. Para descargarlo, basta con visitar la web de Android [32] y asegurarse de acceder a la última versión publicada (durante la redacción de estas líneas, la última versión es la 1.0 de septiembre de 2008).

Una vez descargado el SDK, es necesario descomprimirlo. La ubicación de los ficheros resultantes no es relevante, pero conviene recordar la ruta para pasos posteriores.

### 3.10.2 Descargar Eclipse Ganymede

La descarga de Eclipse no es muy diferente al SDK de Android. La web de Eclipse [33] ofrece multitud de versiones de este entorno de desarrollo según las necesidades del desarrollador. En este caso, es suficiente con obtener la versión 3.4, denominada Ganymede.

Finalizada la descarga, no se realiza ningún proceso de instalación; simplemente se debe descomprimir los ficheros y pulsar el ejecutable para abrir la aplicación. La primera vez que se inicie Eclipse, pide al usuario una localización para el *workspace*, donde se ubicarán por defecto todos los proyectos desarrollados.

### 3.10.3 Instalar el plug-in de Android

El siguiente paso consisten en instalar un *plug-in* específico de Android para la plataforma Eclipse. Esta herramienta, llamada ADT (Android Development Tools), facilita enormemente la creación de proyectos, su implementación, depuración y ejecución, por lo que es altamente recomendable si se quiere trabajar con Android.

Para instalar el *plug-in* ADT en Eclipse Ganymede, es necesario seguir las siguientes indicaciones:

1. Iniciar Eclipse

2. Seleccionar la pestaña *Help* > *Software Updates*. Esta acción abrirá una nueva ventana llamada *Software Updates and Add-ons*.

3. Pinchar en la pestaña *Available Software* y pulsar el botón *Add Site*.

4. Introducir la siguiente URL y pulsar *OK*:

```
https://dl-ssl.google.com/android/eclipse/
```

5. Volviendo a la ventana *Software Updates and Add-ons*, marcar la casilla correspondiente a *Developer Tools* y pulsar el botón *Install*. Se abrirá una nueva ventana.

6. Cerciorarse de que las opciones *Android Developer Tools* y *Android Editors* están marcadas y pulsar el botón *Finish*.

El proceso de instalación dará comienzo y puede llevar algunos minutos. Con el fin de que los cambios tengan efecto, es necesario reiniciar Eclipse.

### 3.10.4 Referenciar el SDK de Android

Tras abrir de nuevo Eclipse, debe indicarse en las preferencias de Eclipse la localización del SDK a utilizar para los proyectos de Android:

1. Seleccionar la pestaña *Window* > *Preferences*, lo que abrirá una nueva ventana.

2. Elegir *Android* en el panel izquierdo.

3. Pulsar el botón *Browse* e indicar la ruta del SDK de Android.

4. Pulsar el botón *Apply* y después *OK*.

### 3.10.5 Actualizaciones del plug-in

Es posible que con el tiempo haya disponible una nueva versión del ADT correspondiente a la versión del SDK de Android instalado. Para comprobar las posibles actualizaciones a través de Eclipse, se debe realizar lo siguiente:

1. Iniciar Eclipse.

2. Seleccionar la pestaña *Help* > *Software Updates*. Esta acción abrirá una nueva ventana llamada *Software Updates and Add-ons*.

3. Elegir la pestaña *Installed Software*.

4. Pulsar el botón *Update*.

5. Si existe alguna actualización para el ADT, seleccionarla y pulsar el botón *Finish*.

## *3.11 Primera aplicación en Android: "Hola Mundo"*

La mejor forma de empezar a comprender las características de cualquier nuevo sistema o entorno de desarrollo es mediante la creación de un ejemplo sencillo. En este caso, y como viene siendo habitual, se creará una aplicación que mostrará por pantalla el mensaje "Hola Mundo".

Además de ilustrar el funcionamiento de Android, se podrá comprobar la utilidad de disponer de una herramienta como el ADT para Eclipse y se explicarán los elementos que componen un proyecto para Android.

### 3.11.1 Crear un nuevo proyecto

La creación de un nuevo proyecto de Android no representa mayor dificultad que crear cualquier otro tipo de proyecto en Eclipse. Una vez abierto este entorno de desarrollo, deben realizarse los siguientes pasos:

1. **Crear el proyecto**
   Seleccionar la pestaña *File* > *New* > *Project*, y en el menú resultante desplegar la opción *Android* y elegir *Android Project*. Pulsar el botón *Next*.

2. **Especificar las propiedades**
   A continuación se muestra una ventana donde es necesario detallar algunas propiedades del proyecto. En concreto, se precisa asignar un valor para:

   - *Project name*: es el nombre del proyecto. En la práctica, será el nombre que reciba la carpeta donde se guardará todo lo relativo al presente proyecto, dentro del *workspace*. Por ejemplo, "HolaMundo".

   - *Package name*: el nombre del paquete bajo el cual será desarrollado todo el código. Por ejemplo, se le puede asignar el valor "com.android.hola".

   - *Activity name*: es el nombre de la clase `Activity` que será creada de forma automática por el *plug-in*. Esta clase `Activity` simplemente es una clase ejecutable, capaz de realizar alguna tarea, y es imprescindible en la mayoría de las aplicaciones para Android. Por ejemplo, póngase el nombre "HolaMundo".

   - *Application name*: el nombre de la aplicación que se va a desarrollar. Constituye el nombre visible para el usuario del dispositivo móvil. Por ejemplo, "Saludo al Mundo".

Tras pulsar el botón *Finish*, Eclipse mostrará en el explorador de paquetes los elementos que conforman el actual proyecto. Además, si se visita el *workspace* asignado para los

proyectos de Eclipse, podrá observarse que existe una nueva carpeta denominada "HolaMundo". El contenido de esta carpeta se explica más adelante.

### 3.11.2 Añadir una interfaz de usuario

Para continuar con el ejemplo, debe desplegarse en el explorador de paquetes la carpeta denominad "src". En esta carpeta existirán dos ficheros: "HolaMundo.java" y "R.java". Ambos constituyen hasta el momento los dos únicos ficheros fuente del proyecto.

Si se abre el fichero "HolaMundo.java", puede encontrarse el siguiente código:

```
package com.android.hola;

import android.app.Activity;
import android.os.Bundle;

public class HolaMundo extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

**Código 5. Código inicial de "HolaMundo.java"**

Este código es perfectamente ejecutable, pero para este ejemplo se van a añadir algunos cambios. El objetivo es crear una simple interfaz de usuario que permita mostrar el mensaje "Hola Mundo". El nuevo código resultante ha de ser el siguiente:

```
package com.android.hola;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HolaMundo extends Activity {

   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);

        tv.setText("Hola Mundo");
        setContentView(tv);
    }
}
```

**Código 6. Código de HolaMundo.java modificado**

En Android, las interfaces de usuario están formadas por una jerarquía de clases llamadas Views (vistas). Una vista o View representa cualquier elemento que se pueda dibujar en pantalla, como un botón, una animación, una etiqueta de texto, etc. En este ejemplo se crea una etiqueta de texto mediante una clase denominada `TextView`.

Al crear una instancia de la clase `TextView`, es necesario pasarle al constructor un determinado contexto de Android. Un contexto representa una especie de manejador del sistema, que facilita la resolución de recursos, el acceso a bases de datos o la configuración de ciertas características de la aplicación. La clase `Activity` hereda de la clase `Context`, por lo que la clase HolaMundo también es un contexto válido en Android.

```
        TextView tv = new TextView(this);
```

**Código 7. Creación de un objeto TextView**

Una vez creada la etiqueta de texto, para asignarle un valor simplemente se llama al método `setText()` con la cadena "Hola Mundo" como argumento.

```
        tv.setText("Hola Mundo");
```

**Código 8. Especificación del mensaje para el objeto TextView**

Como paso final, deben vincularse los elementos visuales que se hayan creado (las "vistas", como `TextView`) con la pantalla que la aplicación mostrará al usuario. Este paso se realiza mediante el método `setContentView()` de la clase `Activity`, que asigna una determinada vista indicando al sistema que es la que debe ser la mostrada por la pantalla.

```
        setContentView(tv);
```

**Código 9. Establecer la vista para el usuario**

### 3.11.3 Ejecutar la aplicación

Antes de lanzar y probar la aplicación, debe crearse una configuración específica para la ejecución. En esta configuración se debe indicar el proyecto a ejecutar, la *Activity* que iniciará la ejecución, las opciones del emulador y otros valores opcionales.

1. Seleccione la pestaña *Run* > *Run Configurations*. Se abrirá una nueva ventana.

2. En el panel de la izquierda, hay que localizar *Android Application* y pulsar el icono con la leyenda *New launch configuration*.

3. Asignar un nombre para esta configuración. Por ejemplo, "Hola Mundo Conf".

4. En la etiqueta *Android*, pulsar el botón *Browse* para localizar el proyecto "HolaMundo" y seleccionar la opción *Launch* para localizar la actividad "com.android.hola.HolaMundo".

5. Pulsar el botón *Run*.

Esto lanzará el emulador de Android. La primera vez que se lanza, puede tardar unos segundos en arrancar. Es recomendable no cerrar el emulador mientras se esté editando y probando un proyecto, para evitar así tener que repetir siempre el proceso de carga.

Si se muestra un error parecido a "*Cannot create data directory*", es necesario acudir a *Run Configurations* > pestaña *Target*, y buscar el campo *Additional Emulator Command Line Optiones*. En dicho campo, introducir la orden "-datadir C:\" si, por ejemplo, se desea crear los ficheros imagen del emulador en el directorio C:\.

La apariencia del emulador debe ser algo similar a la Figura 12:


**Figura 12. Ejecución en el emulador de "Hola Mundo"**

## 3.12 Contenido de un proyecto Android

Al crear un proyecto Android, en el *workspace* de Eclipse se genera una nueva carpeta con el nombre de dicho proyecto. Esta carpeta contiene una serie de subcarpetas y de ficheros que constituyen la anatomía completa de un proyecto Android. En este apartado se intentará dar una visión general del significado y cometido de cada uno de estos elementos.

Volviendo al ejemplo "Hola Mundo", si se echa un vistazo al *workspace* se podrá observar una jerarquía similar a la mostrada en la Figura 13:


**Figura 13. Jerarquía en el workspace de Eclipse**

Las carpetas y ficheros de "HolaMundo" han sido creadas por el *plug-in* ADT, y sus características dependerán siempre del tipo de proyecto que se esté construyendo. Los principales elementos que podemos encontrar se describen a continuación.

### 3.12.1 Carpeta \src

Carpeta que contiene los archivos fuente .java del proyecto. Utiliza la misma jerarquía de carpetas que la indicada por el nombre del paquete que se haya asignado. Por ejemplo, en el caso de la aplicación "HolaMundo" se especificó el nombre de paquete "com.android.hola", por lo que se podrá encontrar la siguiente jerarquía:

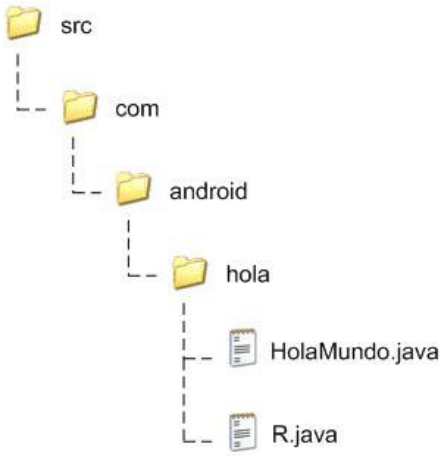


**Figura 14. Jerarquía de la carpeta "src"**

La última carpeta tiene los ficheros fuente en java. El primero de ellos, "HolaMundo.java", aloja la única clase que se creó para la aplicación; el segundo fichero, "R.java", es un archivo que siempre se adjunta por defecto a cualquier proyecto Android y que declara una serie de índices o referencias a los recursos externos que se utilizan en el proyecto actual.

### 3.12.2 Carpeta \res

La carpeta "\res" alberga los recursos utilizados en el proyecto. Por recurso se entiende cualquier fichero externo que contenga datos o descripciones referentes a la aplicación, y que debe ser compilado junto a los ficheros fuente. Esta compilación permite al recurso ser accedido de forma más rápida y eficiente.

Android contempla muchos tipos de recursos, como XML, JPEG o PNG, entre otros. En la mayoría de los casos, un recurso constituye (o viene acompañado de) un fichero XML que lo describe y/o configura. Por ejemplo, un caso típico de recurso en Android son las traducciones de una aplicación a distintos idiomas. A pesar de ser cadenas de texto, se representan como recurso a través de un documento XML.

Cada uno de los recursos utilizados en una aplicación para Android ha de estar localizado en la carpeta adecuada, en función de su naturaleza. De esta forma, dentro de la carpeta de recursos "\res" se pueden encontrar las siguientes subcarpetas:

- `\anim`: aquí se ubican aquellos ficheros XML que describen una animación para un determinado elemento. Las animaciones son movimientos y efectos gráficos básicos.

- `\drawable`: esta carpeta contiene recursos que pueden ser dibujados en la pantalla. Por ejemplo, imágenes con formato JPG, PING o GIF.

- `\layout`: que contiene layouts o diseños que pueden ser usadas para construir interfaces. Un diseño, que representa la pantalla completa o simplemente una parte, se describe a través de un fichero XML.

- `\values`: esta carpeta tendrá ficheros XML que declaran valores de diferentes tipos. Por ejemplo, cadenas de texto, colores predefinidos, arrays de elementos, dimensiones, estilos, etc. En cierta medida, los recursos aquí localizados pueden verse como declaraciones de variables que serán accedidas después desde el código. Por convención, existirá dentro de esta carpeta un fichero XML por cada tipo distinto de recurso que se declare: "strings.xml" para los *strings*, "colors.xml" para los colores, "dimens.xml" para las dimensiones, "integers.xml" para los valores enteros, etc.

- `\xml`: donde se ubican aquellos ficheros XML genéricos que pueden ser procesados como tales desde el código (es decir, utilizando un parseador de XML).

- `\raw`: en esta carpeta se incluyen aquellos recursos que son añadidos a la compilación directamente sin procesar, de forma que la aplicación podrá leerlos como flujos de bytes. Un recurso típico de esta carpeta son los archivos de audio.

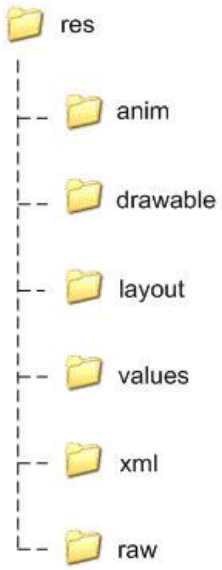


**Figura 15. Jerarquía de la carpeta "res"**

Todos los recursos que aquí se declaren quedarán reflejados en el fichero fuente "R.java". Este fichero se genera y sincroniza de forma automática en tiempo de compilación según el contenido de la carpeta "\res", por lo que está completamente desaconsejado realizar manualmente cualquier tipo de cambio. Puede forzarse la sincronización en Eclipse utilizando la opción *Refresh* del proyecto. El fichero "R.java" contendrá una subclase por cada tipo distinto de recurso que se haya declarado. A su vez, estas subclases declaran una serie de identificadores que representan, uno por uno, cada uno de los de recursos declarados de ese tipo concreto.

Gracias a este fichero, los recursos se pueden utilizar y referenciar directamente en el código de la aplicación, como si de variables globales se trataran. Existen varios mecanismos para invocar estos recursos en el código de la aplicación, pero el más genérico de ellos es mediante el método `getResources()` de la clase `Context`.

### 3.12.3 Carpeta \bin

Esta carpeta contiene los archivos binarios del proyecto generados a partir de los archivos fuente. Al igual que la carpeta "\src", se mantiene la misma jerarquía de subcarpetas que la representada en el nombre del paquete.

### 3.12.4 Archivo AndroidManifest.xml

Un elemento imprescindible en cualquier aplicación para Android es el archivo denominado "AndroidManifest.xml". Este archivo, generado de forma automática por el *plug-in* de Eclipse, representa un manifiesto escrito en XML que describe de forma genérica cada uno de los componentes que forman la aplicación (a saber: *Activity*, *Broadcast Intent Receiver*, *Service* y *Content Provider*). Esta descripción abarca aspectos como sus capacidades y requisitos, las clases que los implementan, los datos que pueden manejar o cuándo deben ser lanzados. En este fichero también pueden concretarse permisos y políticas de seguridad que afectan a toda la aplicación.

**Formato del manifiesto**

La estructura que ha de tener este importante documento se muestra a continuación a través de un básico DTD. No se han incluido en él los atributos que pueden llevar algunos elementos, ya que excedería el objetivo ilustrativo que persigue, pero sí se indica su cardinalidad:

```
<!ELEMENT manifest
      (uses-permission*, permission*, instrumentation*, application?)>
<!ELEMENT application
      (activity+, receiver*, service*, provider*)>
<!ELEMENT activity
      (intent-filter*, metadata*)>
<!ELEMENT receiver
      (intent-filter*, metadata*)>
<!ELEMENT service
      (intent-filter*, metadata*)>
<!ELEMENT provider
      (metadata*)>
<!ELEMENT intent-filter
      (action+, category+, data*)>
```

**Código 10. DTD reducido de un fichero "AndroidManifest.xml"**

Los elementos XML que se pueden encontrar en la declaración del manifiesto se describen en las siguientes líneas:

- **`<manifest>`**: el nodo raíz, bajo el cuál se declararán todos los contenidos del manifiesto.
- **`<uses-permission>`**: declara requisitos de seguridad para que el paquete pueda ser desplegado correctamente.

- **<permission>**: define permisos utilizados para restringir a los componentes de esta aplicación el acceso a ciertos servicios de Android.

- **<instrumentation>**: declara componentes utilizados para probar la funcionalidad de este u otro paquete.

- **<application>**: elemento raíz que enumera, uno por uno, los componentes básicos que constituyen la presente aplicación. Contempla muchos atributos que matizan su configuración.

- **<activity>**: representa un componente *Activity* presente en la aplicación, encargado de representar una acción concreta y generalmente asociado a una interfaz de usuario. Deberá haber tantos elementos `<activity>` como componentes *Activity* haya en la aplicación.

- **<intent-filter>**: declara lo que se denomina un *Intent Filter*, encargado de describir cuándo y dónde puede ejecutarse la *Activity* de la que pende. Mediante este elemento se especifica qué acciones puede manejar la presente actividad, es decir, qué elemento *Intent* puede atender. Permite además la inclusión de varios atributos que matizan su funcionamiento.

- **<action>**: acción soportada por la *Activity*.

- **<category>**: categoría de la *Activity*.

- **<data>**: datos aceptados por la *Activity*.

- **<meta-data>**: metadatos sobre la *Activity* definidos por el desarrollador.

- **<receiver>**: elemento que representa a un componente *Broadcast Intent Receiver*, cuya misión es lanzar una acción en respuesta a un evento. Deberá haber tantos elementos `<receiver>` declarados como componentes *Broadcast Intent Receiver* haya en la aplicación.

- **<service>**: este elemento se corresponde con un componente *Service* presente en la aplicación, encargado de ejecutar una acción en *background*. Deberán aparecer tantos elementos `<service>` como componentes *Service* haya en la aplicación.

- **<provider>**: elemento que encarna a un componente *Content Provider*, utilizado en la aplicación para almacenar y compartir datos. Debería haber tantos elementos `<provider>` declarados como componentes *Content Provider* haya en la aplicación.

Un elemento especialmente interesante dentro del fichero "AndroidManifest.xml" y que merece la pena extender un poco más es `<intent-filter>`. Tal y como se expone en

el listado anterior, este elemento especifica cuándo y dónde puede ejecutarse una determinada *Activity*.

En Android, cuando una aplicación está ejecutándose, quiere realizar una determinada acción y necesita para ello algún componente externo entre los disponibles (como, por ejemplo, abrir una página web o utilizar un visor de imágenes) lanza lo que se denomina un *Intent*. Este elemento contiene información sobre quién desea hacer algo, qué desea hacer y cómo desea hacerlo. Es entonces cuando Android utiliza los *Intent Filter* declarados en los "AndroidManifest.xml" para filtrar y decidir: compara en cada uno de ellos el valor del elemento `<intent-filter>` y averigua qué aplicación es la más óptima para hacerse cargo de la acción requerida.

### Manifiesto de "Hola Mundo"

En el ejemplo anteriormente visto, "Hola Mundo", se puede encontrar el siguiente "AndroidManifest.xml", generado de forma automática por el *plug-in* de Eclipse:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.hola">

<application android:icon="@drawable/icon"
      android:label="@string/app_name">

<activity android:name=".HolaMundo" android:label="@string/app_name">
       <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
       </intent-filter>
</activity>

</application>

</manifest>
```

**Código 11. Manifiesto del ejemplo "Hola Mundo"**

En este manifiesto, primero se declara el documento como un XML válido y se adjunta un espacio de nombres adecuado en el elemento `<manifest>`, así como el paquete al que pertenece con el atributo `package`.

A continuación, existe un elemento `<application>` que se dispone a declarar los componentes que forman la aplicación completa. Recuérdese que solamente puede haber un único elemento `<application>` por manifiesto. El atributo `android:icon` indica que existe un recurso del tipo "drawable", ubicado en la carpeta HolaMundo\res\drawable y de nombre "icon", que debe ser usado como el icono

representativo de esta aplicación. Además, el atributo `android:label` señala que existe también un recurso, de tipo *string* y con nombre "app_name", que será la etiqueta de texto visible para el usuario. Este recurso se encontrará ubicado en la carpeta HolaMundo\res\values, dentro del fichero "strings.xml".

Dentro del elemento `<application>` se declara el único componente que constituye la aplicación, un componente *Activity*. El elemento `<activity>` declara un atributo `androide:name` que especifica el nombre de la clase que implementa dicha *Activity*, así como la misma etiqueta de texto "app_name" para darle un nombre visible, a través del atributo `android:label`.

Por último, la *Activity* declarada lleva asociado un *Intent Filter*, que anuncia cómo y cuándo debe ser lanzada dicha actividad. El elemento `<intent-filter>` define, mediante los elementos `<action>` y `<category>`, que la presente actividad ha de ser considerada la clase principal de la aplicación (`android.intent.action.MAIN`) y que se ha de ejecutar cuando el usuario quiera lanzar la aplicación "Hola Mundo" (`android.intent.category.LAUNCHER`).

## 3.13 Definición de interfaces de usuario con XML

En el ejemplo de "Hola Mundo", se crea una sencilla interfaz consistente únicamente en una ventana con un título desde la que se imprime un mensaje. Esta interfaz está embebida en el propio código fuente; en casos tan sencillos como éste, cambiar la interfaz para ampliarla o sencillamente para darle un aspecto completamente nuevo a nuestra aplicación sería tarea fácil.

Sin embargo, piénsese ahora en aplicaciones para Android mucho más complejas, con cientos o miles de líneas de código fuente dedicadas únicamente a la interfaces de usuario que pueda tener. Este código acaba siempre vinculándose en exceso al resto de código fuente ajeno a la interfaz, provocando que ligeros cambios de interfaz signifiquen muchas horas de trabajo en cambios y nuevos diseños del código completo. Por ello, Android contempla una forma más sencilla, y sobre todo independiente, de configurar interfaces de usuario sin tener que afectar apenas al código fuente de la aplicación.

El método alternativo para construir interfaces en Android consiste en utilizar ficheros escritos en XML y configurar en ellos los aspectos relacionados con esas interfaces. El fichero XML resultante se vincula al proyecto simplemente ubicándolo en la carpeta de recursos "\res\layout" y referenciándolo brevemente en el código fuente.

Por ejemplo, una forma de utilizar XML para crear la misma interfaz de usuario que ya se tenía en "Hola Mundo" sería la siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:text="Hello, Android"/>
```

**Código 12. Ejemplo de interfaz con XML**

El formato del fichero es muy simple: consiste en construir un árbol de elementos, de tal forma que cada elemento se corresponda con el nombre de una clase `View` válida (como `TextView`) y utilizar los atributos y la anidación de elementos para configurar el tipo de interfaz que se desee.

En el Código 12 se puede comprobar como se define, en primer lugar, el documento XML como un documento válido. A continuación, se indica un único elemento `<TextView>`, que se corresponde con la clase `TextView` utilizada ya anteriormente. Observamos los siguientes atributos:

- `xmlns:android`: declara el espacio de nombres Android válido.
- `android:layout_width`: indica qué cantidad del ancho de la pantalla ha de ser ocupado por este elemento View. En este caso, se declara que la caja de texto ocupe toda la pantalla.
- `android:layout_height`: declara qué cantidad del alto de la pantalla va a ocupar el elemento View al que hace referencia. Al igual que antes, en este caso se desea que ocupe todo el alto de la pantalla.
- `android:text`: simplemente indica el texto que se desea mostrar en este elemento View.

Para comprobar su funcionamiento, se deben realizar dos pasos:

1. Sustituir el contenido del fichero "main.xml" presente en la carpeta "\res\layout", por el XML presente en el Código 12.

2. Cambiar el fichero fuente "HolaMundo.java" para que tenga contenido mostrado en el Código 13:

```
package com.android.hola;

import android.app.Activity;
import android.os.Bundle;

public class HolaMundo extends Activity {
```

```
 /** Called when the activity is first created. */
     @Override
public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.main);
     }
 }
```

**Código 13. Código fuente de "Hola Mundo" usando interfaz en XML**

El único cambio presente en "HolaMundo.java" es la eliminación de las referencias a la clase `TextView`, sustituyéndolas por una única llamada a `setContentView()`. Este método accede al diseño o layout llamado "main" presente en el fichero de recursos "R.java". En este, como se recordará, quedan declarados de forma completamente automática, y sin intervención del desarrollador, todos los tipos de recursos existentes en la carpeta "\res".

Más adelante, en esta misma memoria, se detallarán los aspectos concernientes a la construcción de interfaces de usuario más complejas utilizando declaraciones en formato XML.

# 4 CONTACTMAP: LOCALIZADOR DE CONTACTOS

En este capítulo se explica paso a paso el desarrollo completo de *ContactMap*, una aplicación cuyo objetivo es localizar y mostrar en un mapa los contactos almacenados en el dispositivo, añadiendo además otra serie de funcionalidades.

Mediante este desarrollo se busca ilustrar de una forma más práctica las características principales que ofrece Android y pretende además servir como ejemplo para la creación de otras aplicaciones.

Algunos de los aspectos más interesantes de Android y que se explican con esta aplicación son los siguientes:

- Uso de componentes *Activity*.
- Uso de componentes *Service*.
- Solicitudes a través de *Intents*.
- Utilización de los servicios de Google Maps.
- Comunicaciones por HTTP.
- Uso de la base de datos SQLite.
- Composición del archivo "AndroidManifest.xml".
- Acceso a la información de los contactos.
- Acceso a información sobre el propio dispositivo.
- Uso de interfaces remotas con AIDL.
- Composición de interfaces de usuario, tanto con código como con XML.
- Declaración y uso de recursos externos.
- Composición gráfica de elementos en pantalla.
- Gestión de opciones de menú.
- Control del GPS.
- Control de la conexión Wi-Fi.
- Llamadas telefónicas.
- Envío de SMS.
- Envío de correo electrónico.

## *4.1 Análisis y diseño de la aplicación*

### 4.1.1 Introducción a ContactMap

El objetivo básico de *ContactMap* es mostrar al usuario la ubicación geográfica de los contactos que tenga almacenados en el propio dispositivo móvil. Para ello utiliza fundamentalmente tres servicios:

- Google Maps, para mostrar al usuario las localizaciones de los contactos
- La señal GPS del propio dispositivo móvil, para conocer la propia localización
- Una conexión a Internet, con el fin de poder intercambiar información de localización con los demás usuarios.

El usuario visualiza la situación geográfica de sus contactos gracias a los **mapas y servicios de Google Maps**, que son ofrecidos por Android a través de una serie de API específicas. El uso de esta potente herramienta permite tener acceso a miles de imágenes y mapas, convirtiéndose en uno de los elementos de Android que más juego está dando en todo tipo de aplicaciones dentro de la recién creada comunidad de desarrolladores.

Así mismo, la aplicación utiliza el **componente GPS** presente en el mismo dispositivo móvil para conocer la propia ubicación, pudiendo de este modo tanto mostrársela al usuario en el mapa como comunicársela al resto de usuarios. El servicio de GPS, aunque todavía hoy no es de los accesorios más extendidos en los dispositivos móviles, sí que lo será a corto plazo al estar incluido en muchos de los nuevos modelos de *smartphones* a la venta (el primer teléfono con Android, el modelo HTC G1, incluye GPS de serie) y todo parece indicar que en pocos años su incorporación será tan frecuente como lo es hoy la cámara digital o el reproductor de música.

*ContactMap* no requiere la presencia de GPS para poder funcionar, pero su ausencia significa que el usuario no podrá conocer su propia ubicación ni, por tanto, comunicársela a los demás usuarios.

Por otro lado, mediante el establecimiento de una **conexión a Internet** a través de la Wi-Fi, la aplicación *ContactMap* realiza intercambios de información sobre localizaciones con los demás usuarios. Este intercambio no se realiza nodo a nodo, es decir, entre cada usuario, sino que existe un servidor central donde cada uno de los usuarios consulta las localizaciones de sus contactos, además de actualizar su propia localización. Este intercambio se realiza utilizando el estándar XML y sobre el protocolo HTTP, aunque el funcionamiento mas detallado de esta comunicación y el del servidor se detalla en apartados posteriores.

Al igual que con el GPS, no es imprescindible disponer de manera constante de una conexión a Internet para que *ContactMap* pueda mostrar las ubicaciones de los contactos. Por defecto, *ContactMap* intenta mostrar la ubicación más reciente de cada usuario, pero en caso de no poder conectarse mostrará siempre la última ubicación conocida. Por ello sí es necesario conectarse al menos una vez para obtener las localizaciones correspondientes

Además de mostrar ubicaciones geográficas, *ContactMap* permite también realizar otra serie de acciones asociadas a cada uno de los contactos como, por ejemplo, realizar una llamada telefónica, enviar un mensaje de texto o escribir un correo electrónico.

### 4.1.2 Casos de uso

En este apartado se utilizan los casos de uso como forma de acercar al lector las funcionalidades que la aplicación *ContactMap* va a desempeñar frente al usuario. Sin embargo, no se va a profundizar demasiado en las posibilidades que ofrece esta técnica, sino que se limitará solamente en aquellos aspectos que son más ilustrativos y concisos para el lector y que le ayuden a comprender mejor qué es lo que realiza la aplicación.

Como recordatorio se dirá que, en ingeniería del software, un caso de uso representa un uso típico que se le da al sistema. La técnica de los casos de uso permite capturar y definir los requisitos que debe cumplir una aplicación, y describe las típicas interacciones que hay entre el usuario y esta. Dicha técnica es utilizada con frecuencia por los ingenieros del software para, entre otras cosas, mostrar al cliente de forma clara y sencilla qué tipo de acciones podrá realizar su futuro sistema.

A continuación se muestra un diagrama con los casos de uso asociados a *ContactMap*, utilizando el estándar UML 2.0 [35].

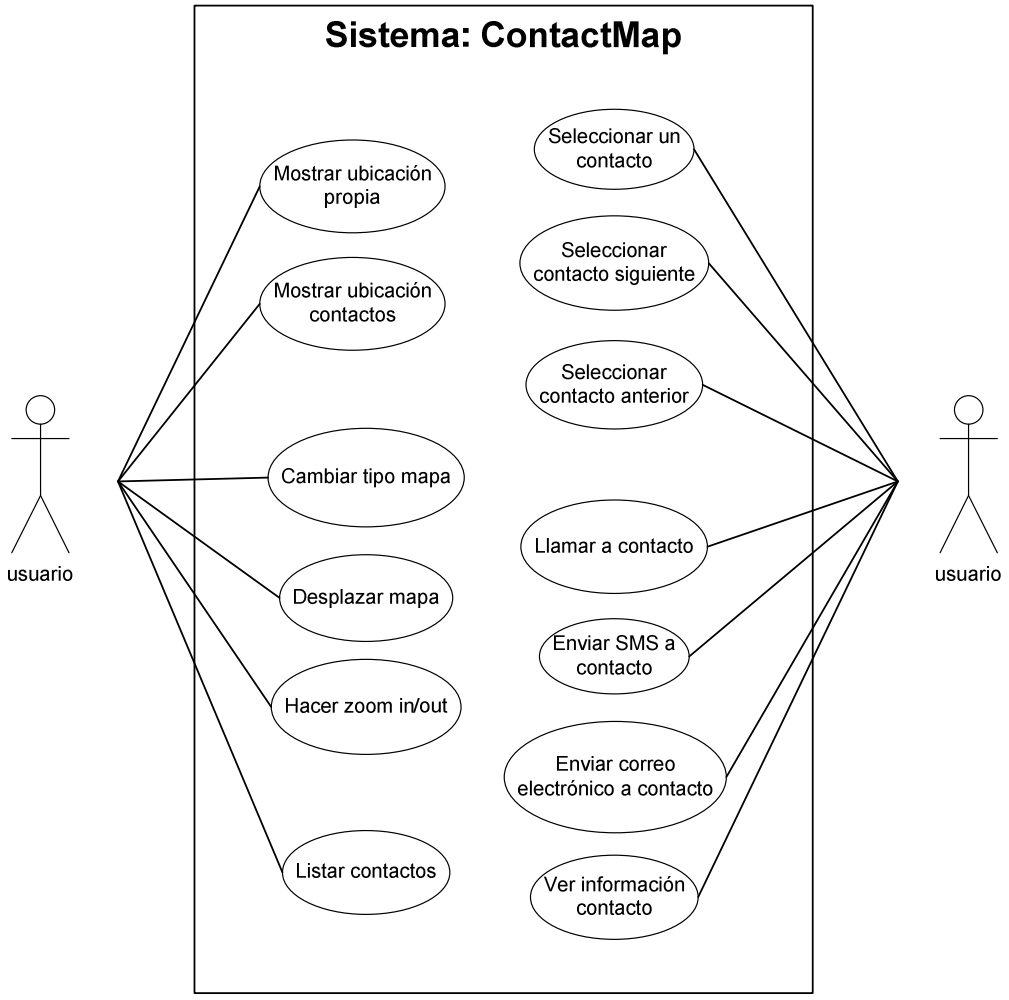


**Figura 16. Diagrama de casos de uso de *ContactMap*.**

En diagrama mostrado en la Figura 16, el sistema, (es decir, la aplicación *ContactMap*) está representado por una caja que contiene los casos de uso. Cada caso de uso consiste en un óvalo con un nombre descriptivo en su interior. Fuera del sistema se encuentra los actores que pueden interactuar con él. En este caso, a pesar de estar representado por dos figuras por motivos de espacio y legibilidad, existe un **único actor** de nombre "usuario" que es el que realiza todos los casos de uso.

A continuación se ofrece una breve descripción del cometido de cada uno de los casos de uso mostrados en el diagrama:

- **Mostrar ubicación propia**: mostrar en el mapa la ubicación actual del usuario.

- **Mostrar ubicación contactos**: mostrar en el mapa las ubicaciones de los contactos del usuario, siempre y cuando se disponga de información de localización.

- **Cambiar tipo de mapa**: alternar la imagen entre la vista de satélite (fotografía aérea) y vista de callejero.

- **Desplazar mapa**: desplazar el mapa en el sentido pulsado por el usuario.

- **Hacer zoom in/out**: acerca o alejar el nivel de zoom del mapa.

- **Listar contactos**: listar en orden alfabéticos todos aquellos contactos para los que se disponga de información de localización.

- **Seleccionar un contacto**: elegir un contacto, centrando en él el mapa y haciéndole objeto de las diferentes acciones.

- **Seleccionar contacto siguiente**: elegir al contacto siguiente por orden alfabético al actual, centrando en él el mapa y haciéndole objeto de las diferentes acciones.

- **Seleccionar contacto anterior**: elegir al contacto anterior por orden alfabético al actual, centrando en él el mapa y haciéndole objeto de las diferentes acciones.

- **Llamar a contacto**: realizar una llamada telefónica al contacto actual.

- **Enviar SMS a contacto**: enviar un mensaje de texto al contacto actual.

- **Enviar correo electrónico a contacto**: enviar un correo electrónico al contacto actual.

- **Ver información de contacto**: visualizar la fecha y hora de la última vez que el contacto actualizó su localización.

Siguiendo el mismo criterio, el siguiente diagrama expresa los casos de uso asociados al servidor al que se conecta la aplicación para gestionar la información de localización:

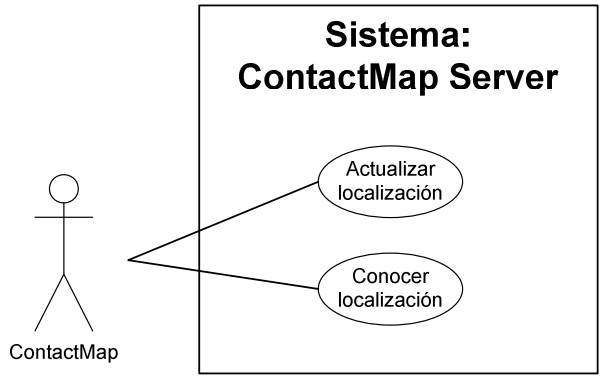


**Figura 17. Diagrama de casos de uso del servidor de *ContactMap***

En el diagrama de la Figura 17, el único actor que realiza los casos de uso es la propia aplicación *ContactMap*, ya que es la que establece la conexión con el servidor. Los casos de uso considerados son:

- **Actualizar localización**: actualizar los datos de localización del usuario.
- **Conocer localización**: obtener los datos de localización del usuario solicitado.

Es importante recalcar que los casos de uso simplemente expresan el punto de vista del usuario sobre cómo debe funcionar la aplicación y qué puede realizar a través de ella, y son una forma de facilitar su comprensión. No tiene porqué existir ninguna correspondencia entre los casos de uso y la clases finalmente implementadas, más allá de que las clases en su conjunto, como sistema completo, realizan aquello que los casos de uso expresan.

### 4.1.3 Intercambio de información con XML

*ContactMap* obtiene la información de localización de los contactos del usuario conectándose a un servidor, donde además actualiza su propia ubicación para que esté disponible para otros usuarios. Estos intercambios de información entre aplicación y servidor se realizan utilizando documentos XML. Como seguramente conocerá el lector, el lenguaje de marcado XML permite expresar de forma sencilla y abierta diferentes estructuras de datos, manteniendo intacta su semántica y contenidos.

El uso de XML entre otras alternativas responde principalmente a dos criterios. Por un lado, se trata de un estándar regulado y ampliamente utilizado, lo que facilita su comprensión y la posible ampliación o interconexión con otros servicios en un futuro. Por otro, es una tecnología muy fácil de usar y cuenta con diferentes API en Java que convierten en algo automático la lectura y composición de este tipo de documentos. En concreto, tanto en el lado servidor como en el lado de la aplicación en Android, se utiliza la biblioteca SAX.

Cuando se desea utilizar documentos XML que cuenten siempre con una misma estructura, como es el caso que nos ocupa, se suelen definir unas plantillas que expresan la forma en la que debe ser construido este documento XML para ser considerado válido. Una de estas plantillas es la que se denomina DTD, y expresa de forma muy clara qué tipo de elementos e información puede contener un determinado documento XML.

En *ContactMap* se definen dos tipos distintos de documentos XML: el de petición y el de respuesta. El siguiente DTD define un documento de petición:

```
<!ELEMENT contactmap-request (me, contact*)>
<!ELEMENT me (number, latitude, longitude, date)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT contact (#PCDATA)>
```

**Código 14. DTD para peticiones**

Un documento de petición consta de dos partes. En la primera, de carácter obligatorio, el usuario notifica sus datos para ser actualizados en el servidor: el número de teléfono que lo identifica, los datos de localización expresados en una latitud y una longitud, y la fecha actual consistente en día, mes, año, hora, minutos y segundos. En la segunda parte del documento de petición, de carácter opcional, se incluyen los números de teléfono de aquellos contactos para los que solicita conocer su ubicación.

El Código 15 muestra un documento de petición válido y aceptado por el servidor. En él, el usuario actualiza sus datos de localización y además solicita la localización de tres de sus contactos:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE contactmap-request SYSTEM "contactmap-request.dtd">

<contactmap-request>

      <me>
            <number>660854377</number>
            <latitude>40.425970</latitude>
            <longitude>-3.696010</longitude>
            <date>2008-01-04 18:30:42</date>
      </me>
```

```
      <contact>647889034</contact>
      <contact>688403241</contact>
      <contact>654312526</contact>

</contactmap-request>
```

**Código 15. Ejemplo de petición XML válida**

Los documentos de respuesta deben, por su parte, cumplir lo definido en la plantilla DTD mostrada en el Código 16:

```
<!ELEMENT contactmap-response (contact+ | error)>
<!ELEMENT contact (number, latitude, longitude, date)>
<!ELEMENT error (#PCDATA)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT date (#PCDATA)>
```

**Código 16. DTD para respuestas**

En el documento de respuesta viajan los datos de localización de aquellos contactos que hayan actualizado en algún momento su ubicación en el servidor. Dicho de otro modo, pueden existir contactos solicitados por la aplicación *ContactMap* en su documento de petición para los cuales no existan datos de localización disponibles, por lo que no serán incluidos en el documento de respuesta. En caso de haber ocurrido algún error no se adjunta información de ningún contacto, sino que se describe el error acaecido.

El siguiente documento de respuesta es un documento válido según la plantilla DTD anteriormente mostrada. En él, se comunican a la aplicación los datos de dos de los tres contactos solicitados según la petición del Código 15, al ser los únicos disponibles. El primer contacto probablemente está utilizando en este momento *ContactMap*, ya que la hora de actualización es casi contemporánea a la comunicada por el usuario de la petición (ver Código 15). El segundo contacto actualizó por última vez su posición el día anterior.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE contactmap-response SYSTEM "contactmap-response.dtd">
<contactmap-response>
```

```
      <contact>
            <number>647889034</number>
            <latitude>40.425968</latitude>
            <longitude>-3.976010</longitude>
            <date>2008-01-04 18:29:15</date>
      </contact>

      <contact>
            <number>688403241</number>
            <latitude>40.428974</latitude>
            <longitude>-3.717503</longitude>
            <date>2008-01-03 11:49:00</date>
      </contact>

</contactmap-response>
```

**Código 17. Ejemplo de respuesta XML válida**

### 4.1.4 Servidor

El servidor al que se conecta *ContactMap* realiza dos funciones principales: actualizar información de localización del usuario y enviarle la información de localización de los usuarios (esto es, sus contactos) solicitados. Mediante la actualización, un usuario cualquier envía sus datos (latitud, longitud y fecha, además de su número de teléfono identificativo) y estos son almacenados en una base de datos. En la consulta, el servidor lee los contactos solicitados y devuelve la información de todos aquellos que estén presentes en esta misma base de datos.

En la composición del servidor funcionan en realidad tres componentes básicos:

- Una base de datos, que almacena toda la información de localización que envían los usuarios.
- Un servlet, que atiende la petición recibida, la procesa y envía la respuesta correspondiente.
- Un servidor web, donde reside y se ejecuta el servlet, y que permanece a la espera de conexiones HTTP entrantes.

Para la **base de datos** se utiliza el gestor MySQL 5.0, usándolo bajo la licencia gratuita que permite su explotación para desarrollos no lucrativos, como es la aplicación que ocupa este proyecto. En él se ha implementado una sencilla base de datos llamada *ContactMapBD* que cuenta con una única tabla; en esta se almacena para cada usuario el número de teléfono que lo identifica, su latitud y longitud, así como la fecha completa de actualización. El siguiente script es el utilizado para generar dicha base de datos.

```
--
-- Crear la base de datos `ContactMapBD`
--
DROP DATABASE `ContactMapBD`;
CREATE DATABASE `ContactMapBD`
DEFAULT CHARACTER SET latin1
COLLATE latin1_spanish_ci;
USE `ContactMapBD`;

-- --------------------------------------------------------
--
-- Estructura de tabla para la única tabla `friend`
--

CREATE TABLE IF NOT EXISTS `friend` (
  `number` bigint NOT NULL,
  `latitude` float (8,6) NOT NULL,
  `longitude` float (8,6) NOT NULL,
  `date` datetime NOT NULL,
  PRIMARY KEY  (`number`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_spanish_ci;
```

**Código 18. Script SQL de la base de datos del servidor**

El **servlet** es el componente que procesa las peticiones que envían los usuarios. Actualiza la información del usuario, accede a la base de datos en busca de los contactos solicitados, y devuelve la respuesta. Como información para el lector, se dirá que un servlet no es más que un componente Java, generalmente pequeño e independiente de la plataforma, que se ejecuta en un servidor web al que extiende su funcionalidad. El servlet implementado recibe el nombre de *ContactMapServlet* y es al que van dirigidas las conexiones.

Cuando recibe una petición, el servlet utiliza la librería SAX de Java para leer el documento XML completo. En primer lugar, actualiza en la base de datos la localización comunicada por el usuario. A continuación, consulta en la base de datos todos aquellos contactos que el usuario ha indicado en la petición recibida. Esta consulta la realiza gracias a las librerías de JDBC y MySQL de Java. Por último, el servlet compone y envía a su vez una respuesta en un nuevo documento XML, donde incluirá los datos de localización de todos aquellos contactos que estén presentes en la base de datos.

Por su parte, el **servidor web** utilizado es Apache Tomcat (también en su distribución gratuita) y es el entorno donde se ejecuta el servlet, quedando siempre a la espera de recibir conexiones HTTP. Todas las comunicaciones entre la aplicación *ContactMap* y el servidor se envían a través de dicho protocolo. En las peticiones, el método HTTP utilizado es POST, correspondiente al envío de datos, y se encapsula el documento

XML en una variable de nombre "xml". En las respuestas, se envía el documento XML correspondiente en el campo de datos.

### 4.1.5 Modelo de clases

Las clases utilizadas para resolver un determinado problema, sus atributos y métodos, la visibilidad que de estos tienen las demás clases, así como las relaciones que existen entre ellas y sus colaboraciones, constituyen el modelo de clases. Mediante este tipo de modelos se expresa, con mayor o menor nivel de detalle, la futura implementación del sistema, así como permite dar una idea bastante cercana a la forma en la que se ha abordado el problema.

En el presente apartado se ofrece al lector una breve descripción de cuál es el modelo de clases utilizado en la aplicación *ContactMap*. Nuevamente, el objetivo principal es ayudar a comprender cómo se soluciona el problema planteado y pretende servir de anticipo a la explicación más detallada de la implementación de *ContactMap*, que se da en secciones posteriores. Por todo ello, el modelo de clases ofrecido no incluye más que clases, atributos, métodos, y la visibilidad de estos, con vistas a no perder al lector con aspectos demasiado complejos.

En la Figura 18 se enseña el diagrama correspondiente al modelo de clases utilizado, siguiendo el estándar UML 2.0 [35].

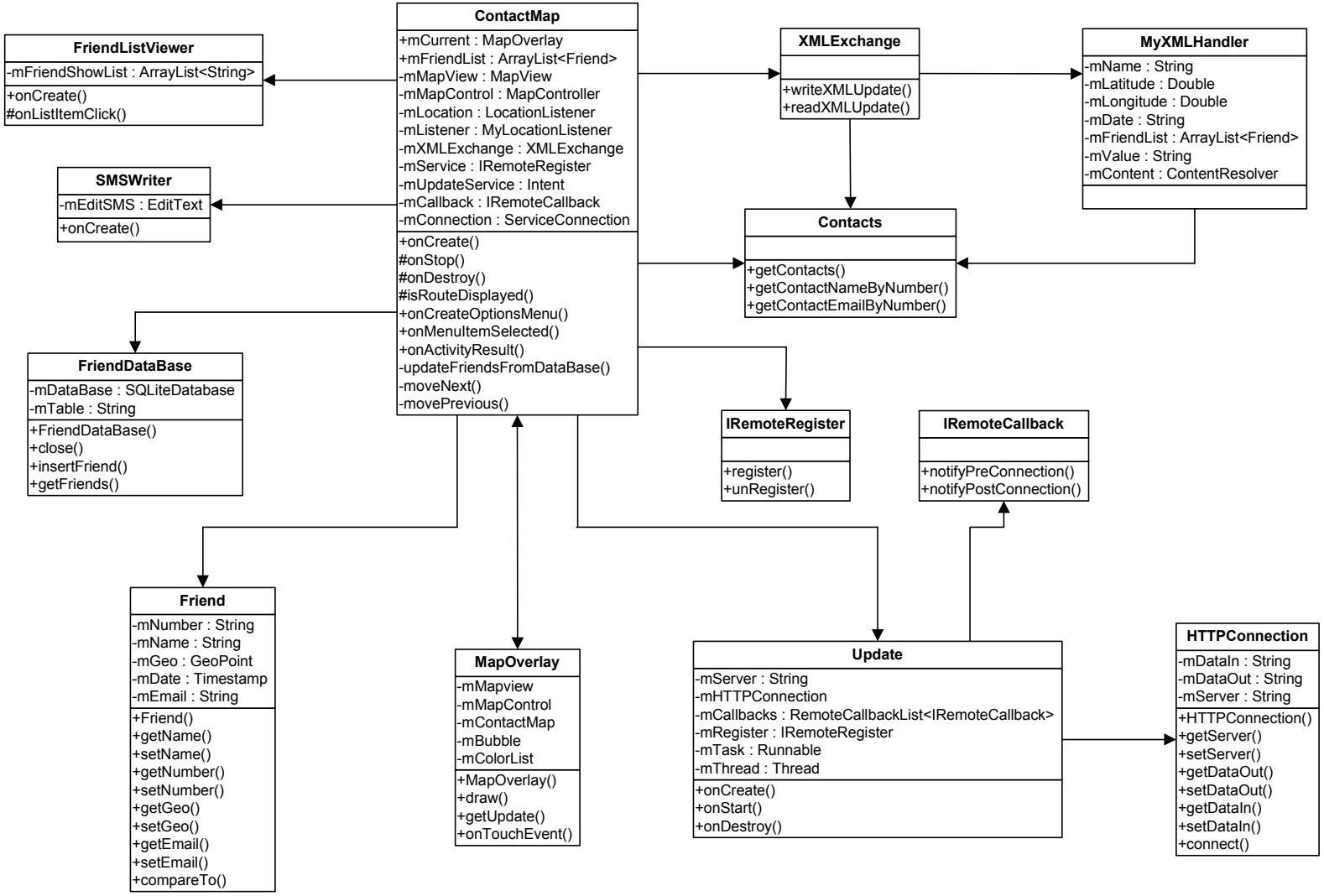


**Figura 18. Diagrama de clases de *ContactMap***

Las clases presentes en el anterior diagrama, así como su naturaleza y cometido principal dentro de la aplicación, se resumen brevemente en las siguientes líneas:

- **`ContactMap`**: es la clase principal y la que coordina la mayor parte de la ejecución de la aplicación. Mediante `ContactMap`, que extiende la clase `Activity` (uno de los componentes básicos de cualquier aplicación para Android), se muestra al usuario el mapa con la ubicación de los contactos, se gestiona la conexión con el servidor, se procesa la información recibida y enviada, así como controla las demás funcionalidades disponibles para el usuario.

- **`MapOverlay`**: esta clase se encarga de los aspectos relacionados con dibujar y actualizar los diferentes elementos de la pantalla, principalmente el mapa y los contactos.

- **`XMLExchange`**: con esta clase se escriben y leen los documentos XML que forman, respectivamente, las peticiones y respuestas del servidor.

- **`MyXMLHandler`**: esta clase ayuda a procesar las respuestas recibidas, construyendo en memoria las estructuras necesarias para representar a los contactos y sus correspondientes localizaciones.

- **`Update`**: es la clase encargada de conectar periódicamente con el servidor para actualizar la información propia y la de los contactos. Extiende la clase `Service`, otro de los componentes básicos de una aplicación Android, ejecutándose siempre en *background*.

- **`HTTPConnection`**: esta clase gestiona los aspectos de más bajo nivel de las conexiones HTTP con el servidor.

- **`Friend`**: la clase que representa a un contacto que puede ser representado en el mapa, ya que se cuenta con su información de localización.

- **`FriendDataBase`**: clase que gestiona el acceso a la base de datos SQLite para almacenar o consultar los datos de localización de los contactos. Se utiliza en caso de no poder conectar con el servidor.

- **`Contacts`**: mediante esta clase se accede a la información de contactos almacenados en el dispositivo móvil.

- **`FriendListViewer`**: esta clase extiende también la clase `Activity`, y muestra una lista de los contactos permitiendo así al usuario seleccionar uno.

- **`SMSWriter`**: clase que extiende la clase `Activity`, y muestra una interfaz con la que el usuario puede escribir un SMS.

- **`IRemoteRegister`**: interfaz remota que permite comunicarse con la clase `Update`, donde está implementada. Se utiliza para comunicar la *Activity* principal (clase `ContactMap`) con el *Service* activo en *background* (clase `Update`).

- **`IRemoteCallback`**: interfaz remota que permite comunicarse con la clase `ContactMap`, donde está implementada. Se utiliza para comunicar el *Service* activo en *background* (clase `Update`) con la *Activity* principal (clase `ContactMap`).

### 4.1.6 Arquitectura

Una vez descrito el funcionamiento general de *ContactMap*, las funcionalidades ofrecidas al usuario, el comportamiento del lado servidor y el intercambio de información mediante XML, a continuación se ofrece un diagrama con la arquitectura general del sistema completo:

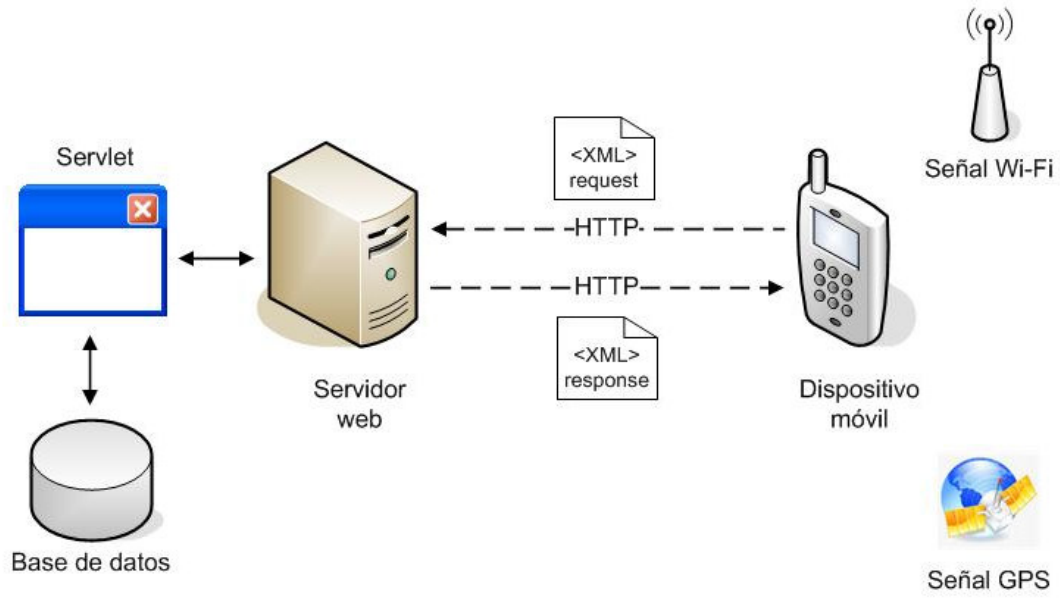


**Figura 19. Arquitectura del sistema completo de *ContactMap***

En la Figura 19 el lector puede apreciar los siguientes elementos:

- **Dispositivo móvil**: dispositivo con el sistema Android corriendo y donde está instalada la aplicación *ContactMap*. Mediante el uso del GPS, el dispositivo conoce la ubicación actual del usuario y a través de la conexión Wi-Fi realiza

intercambios de información con el servidor web: notifica sus datos de localización y solicita la ubicación de sus contactos.

- **Documentos XML**: el dispositivo móvil realiza intercambios de documentos XML con el servidor utilizando el protocolo HTTP. El documento de petición comunica al servidor la ubicación actual del usuario y solicita a su vez las ubicaciones de sus contactos, mientras que el documento de respuesta comunica al dispositivo móvil dichas localizaciones.

- **Servidor web**: permanece siempre a la espera de conexiones por parte de los diferentes usuarios de *ContactMap*. Aloja el servlet que procesa las peticiones y respuestas sobre XML.

- **Servlet**: pequeño programa en Java que recibe peticiones, consulta la base de datos, y construye y envía las respuestas XML de vuelta al dispositivo móvil.

- **Base de datos**: la base de datos almacena toda la información de localización de los usuarios de *ContactMap*. Es consultada por el servlet.

## *4.2 Desarrollo e implementación*

Una vez expuestos los objetivos y características principales de la aplicación, así como las decisiones tomadas en cuanto a su diseño, a continuación se desgranan en este apartado los aspectos relacionados con su implementación.

El fin perseguido en las siguientes líneas no es sólo mostrar el código más relevante de *ContactMap*, sino ofrecer también una explicación general sobre cómo funcionan las aplicaciones para Android, de forma que a su vez se consigan mostrar las peculiaridades de este nuevo sistema. Así mismo, los detalles de implementación mencionados esperan poder ayudar a otros desarrolladores a ampliar en un futuro las capacidades de *ContactMap*, o incluso inspirar la creación de aplicaciones con servicios similares.

Se advierte al lector de que los fragmentos de código fuente a continuación mostrados no son una copia literal del código fuente de *ContactMap*, sino que en algunos casos se reducen o modifican por motivos de limitación de espacio, pero sobre todo por simplificar y facilitar su comprensión.

### 4.2.1 Acceso a Google Maps

La utilización del popular servicio Google Maps es una de las posibilidades más atractivas de Android. En efecto, un gran número de las aplicaciones presentadas al concurso de desarrolladores propuesto por Google utilizan estas bibliotecas con fines muy distintos [36].

El paquete que incluye todas las clases relacionadas con la carga y manejo de mapas directamente desde Google Maps es `com.google.android.maps`. Dentro de este paquete podemos encontrar clases como las siguientes:

- `MapView`: obtiene el mapa solicitado y lo muestra en pantalla.
- `MapController`: gestiona el manejo de un mapa, como desplazamientos o zoom.
- `GeoPoint`: clase que representa un punto geográfico determinado del mapa, según su latitud y longitud.
- `Overlay`: permite dibujar y representar elementos sobre el mapa.
- `MapActivity`: clase muy relevante que extiende la clase base `Activity`, y permite crear una *Activity* específica para gestionar mapas.

Un paso previo a la utilización de este API para mostrar y manejar mapas es el registro en Google Maps y su aceptación de los términos y condiciones de uso. Este registro se realiza a través de una clave denominada en Android como *API key*.

Una vez se ha procedido al registro, la utilización de mapas en nuestra aplicación no requiere más que unas cuantas llamadas a las clases pertinentes. Todo ello se puede comprobar en los dos siguientes apartados.

**Obtención de API Key**

Tal y como se ha mencionado, para utilizar los servicios de Google Maps en una aplicación Android es imprescindible realizar unos pasos previos que implican el registro del desarrollador y la aceptación de las condiciones de uso. De esta forma, Google quiere asegurar que se hará en todo momento un uso adecuado y apropiado de los servicios y datos que nos va proporcionar desde entonces con Google Maps.

Toda aplicación en Android está acompañada de un certificado que asegura su autoría y la vincula con su desarrollador. Si se utiliza el *plug-in* de Android para Eclipse, todas las aplicaciones están firmadas por un cerificado al que se puede calificar de prueba y que permiten a los desarrolladores poder crear y probar sus aplicaciones sin más esperas. Este certificado se obtiene a través del fichero de claves de prueba "debug.keystore", presente por defecto en el SDK de Android.

Recuérdese, sin embargo, que si un desarrollador desea distribuir su aplicación de forma pública o hacerla accesible desde servicios de descarga, como por ejemplo Android Market, es necesario que se cree su propio fichero e individual de claves ".keystore" con el que crear un certificado y firmar su aplicación.

El registro para tener acceso a Google Maps se hace a través de dicho certificado. En concreto el proceso es el siguiente, utilizando el fichero "debug.store". En caso de disponer un fichero de claves propio, el proceso sería el mismo cambiando los parámetros necesarios.

1. En primer lugar, se ha de obtener un resumen MD5 del certificado que se va a usar para firma la aplicación donde se hará uso de mapas. El resumen puede generarse, por ejemplo, con la herramienta *keytool* presente en el SDK de Java. La llamada correspondiente se muestra en la Figura 20.

```
>keytool –list –alias androiddebugkey –keystore
C:\debug.keystore –storepass android –keypass android

Huella digital de certificado (MD5):
0B:41:31:DC:4B:89:22:8E:A5:45:79:C6:13:DA:7E:D4
```

**Figura 20. Ejemplo de uso de la herramienta keytool para generar resumen MD5**

2. A continuación, se debe registrar el resumen MD5 obtenido en Google Maps. Para ello, se visita la web de registro [37], se aceptan los términos de uso, se envía el resumen. Es necesario disponer de una cuenta Google activa.

3. Google Maps proporciona entonces una clave alfanumérica, llamada *API key*. Esta es la que se debe utilizar para obtener y manejar mapas, siendo además única y exclusiva para la aplicación firmada con certificado utilizado.

```
Tu clave es:

0sO8gCzgaeJ3QpNmuPTX3pwzQ7m6vD3_tV2CY4w

Esta clave es válida para todas las aplicaciones firmadas con el
certificado cuya huella dactilar sea:

0B:41:31:DC:4B:89:22:8E:A5:45:79:C6:13:DA:7E:D4

Incluimos un diseño xml de ejemplo para que puedas iniciarte por
los senderos de la creación de mapas:

<com.google.android.maps.MapView
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:apiKey="0sO8gCzgaeJ3QpNmuPTX3pwzQ7m6vD3_tV2CY4w"
/>
```

**Figura 21. Ejemplo de *API key* proporcionada por Google**

### Mostrar un mapa

Una vez se dispone de una *API key* con la que poder acceder a los mapas y servicios de Google Maps, no resta más que hacer las llamadas pertinentes a las clases del paquete `com.google.android.maps`.

La clase `ContactMap`, como se recordará, es la clase principal y la que coordina la mayor parte de las demás clases de la aplicación. Esta clase deriva de la clase `Activity`, como ya se advirtió, pero lo hace de forma indirecta. A quien extiende en realidad es a la clase `MapActivity`, que incluye las funciones estándares de `Activity` más aquellas otras relacionadas con la visión de mapas.

Para poder visualizar un mapa obtenido desde Google Maps, es necesario por lo menos dos objetos básicos:

- Un objeto `MapView`, que obtiene y representa el mapa.
- Un objeto `MapController`, que controla el mapa representado.

Ambos objetos, encargados de contener y controlar el mapa que se está mostrando, se encargan dentro de la clase `ContactMap` a través de dos variables globales, llamadas `mMapView` y `mMapControl`, respectivamente. Ambas variables serán creadas y configuradas inicialmente en esta clase, pero también son accedidas por la clase `MapOverlay`, cuya misión es ir actualizando el mapa con diferentes elementos.

En primer lugar, se crea el objeto `mMapView`. En su constructor es necesario especificar un contexto válido de aplicación (la propia clase `ContactMap`) y además, incluir la *API key* que previamente se ha generado.

El objeto `mMapControl` ha de vincularse a un objeto de la clase `MapView`, para indicar que es en esa vista de mapa donde debe ejercer las labores de control. Para generar el objeto `mMapControl`, se utiliza el método `getController()` del objeto `mMapView`.


**Figura 22. Mapa obtenido desde Google Maps**

Llegados a este punto, ya se tienen los dos elementos básicos, es decir, un mapa y su controlador. Sin embargo, es necesario especificar ahora otros aspectos, como por ejemplo, qué zona se desea mostrar, con qué tipo de mapa o el nivel de zoom.

La clase `GeoPoint`, también incluida en el paquete `com.google.android.maps`, será una de las que más se utilizarán si se trabaja con mapas. Esta clase representa un punto concreto, una localización determinada a través de sus coordenadas de latitud y longitud. Para construir uno de estos objetos, es necesario proporcionar en su constructor dichas coordenadas, siempre multiplicadas por la potencia $10^6$.

Como curiosidad, se mencionan formas rápidas de conocer las coordenadas de un punto concreto:

- En el propio Google Maps, al buscar una dirección se ofrece la posibilidad de generar automáticamente código HTML para publicar esta en una página web. Dentro de este código HTML se encuentra la etiqueta "ll", que contiene el valor de la latitud y longitud, respectivamente.

- Existen algunas páginas web que utilizan Google Maps para buscar e informar al usuario de las coordenadas del punto geográfico indicado, como la indicada en la referencia [38].

Una vez generado un punto mediante coordenadas, es posible centrar el mapa en él. Para llevarlo a cabo, se utiliza el método `setCenter()` de `mMapControl`. Además, en este mismo objeto se tienen los métodos `setSatellite()`, que cambia el tipo de mapa entre vista de satélite o vista de callejero, y `setZoom()` que establece el nivel de zoom deseado para el mapa.

Como paso final para poder generar el mapa, ha de tenerse en cuenta la configuración del "AndroidManifest.xml" de *ContactMap*. En este fichero deben declararse los componentes básicos que actúan, sus características, así como los permisos de usuario. Para poder utilizar mapas, es necesario declarar de forma explícita que se va a utilizar la biblioteca `com.google.android.maps` mediante el elemento `<uses-library>`:

```
<uses-library android:name="com.google.android.maps" />
```

**Código 19. Declaración en el manifiesto del API de Google Maps**

También se ha de conceder a la aplicación permiso para conectarse a Internet y poder así traer los mapas necesarios:

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

**Código 20. Declaración en el manifiesto del permiso de conexión a Internet**

En el Código 21 se muestra el código implementado para mostrar un mapa en pantalla. Solamente se incluye el código relevante para tal fin dentro del método `onCreate()`, que es el método que se llama la primera vez que se crea una *Activity*, en este caso nada más cargar la aplicación.

```
public class ContactMap extends MapActivity {

private MapView mMapView = null;                // Vista del mapa
private MapController mMapControl = null;       // Controlador del mapa
```

```
/** Called when the Activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {


      super.onCreate(savedInstanceState);

      // (...)

      // Crear MapView y MapController asociado al primero
      mMapView = new
      MapView(this,"0sO8gCzgaeJ3QpNmuPTX3pwzQ7m6vD3_tV2CY4w");
      mMapControl = mMapView.getController();

      // Crear un punto
      GeoPoint gPoint = new GeoPoint(40331175,-3765780);

      // Vista satélite
      mMapView.setSatellite(true);
      // Nivel de zoom
      mMapControl.setZoom(16);
      // Centrar el mapa en el punto
      mMapControl.setCenter(gPoint);

      setContentView(mMapView);

      // (...)
}

}
```

**Código 21. Mostrar un mapa de Google Maps**

Se puede comprobar que es necesario indicar a la actividad `ContactMap` qué es lo que ha de mostrar como vista principal al usuario, es decir, el objeto `mMapView` con el mapa cargado. El método `setContentView()` heredado de la clase *`Activity`* realiza esta función.

### 4.2.2 Representación de los contactos en memoria

Para cada contacto, la aplicación necesita conocer su nombre, número de teléfono, su dirección de correo electrónico, sus datos de localización, esto es, latitud y longitud, y la fecha en la que fue actualizada su ubicación. Toda esta información se obtiene desde diferentes fuentes: los contactos almacenados en el dispositivo móvil, el servidor o la base de datos SQLite.

Sin embargo, una vez se han reunido todos estos datos, se representan en memoria a través de una lista llamada `mFriendList` y que es una variable global de la clase `ContactMap`. El reunir todos los datos en una misma estructura, que es la que se

consulta a la hora de representar los contactos en el mapa, permite un acceso mucho más rápido y limpio que si se obtuviera por separado de forma directa desde cada una de las fuentes.

La clase `Friend` representa a un contacto que puede ser mostrado en el mapa, y constituye cada uno de los miembros de la lista `mFriendList`. Tiene cinco atributos declarados:

- `mNumber`, para el número de teléfono,
- `mName`, para el nombre,
- `mEmail`, para la dirección de correo electrónico,
- `mGeo`, para los datos de localización. Es una instancia de la clase `GeoPoint`.
- `mDate`, para la fecha en la que fueron obtenidos los datos de localización.

`mFriendList` se actualiza cada vez que se recibe una respuesta del servidor. La excepción se produce cuando no es posible disponer de Wi-Fi, en cuyo caso se actualiza una única vez desde la base de datos SQLite nada más arrancar la aplicación y detectar tal circunstancia. Este comportamiento se explica en el apartado 4.2.12.

### 4.2.3 Dibujar y gestionar elementos en el mapa

La interfaz principal de *ContactMap* consiste en la vista completa de un mapa donde el usuario visualiza la ubicación de los contactos, puede desplazarlo libremente, realizar cambios en el nivel de zoom, o bien pulsar en un contacto para centrar el foco en él. Además, al tocar sobre el contacto este muestra un bocadillo con información acerca de la fecha en la que notificó su ubicación actual.

Por lo tanto, los elementos que se deben dibujar sobre el mapa visto por el usuario son los siguientes:

- Un punto de color, que representa a un determinado contacto.
- El nombre del contacto.
- Un círculo concéntrico de ese mismo color, en caso de que el contacto sea el que centra el foco de aplicación en ese momento.
- Un bocadillo con la fecha de actualización.

El último elemento dibujable, el bocadillo con información, debe ser pintado en caso de que el usuario pulse sobre el contacto. Por lo tanto, es necesario también tomar el control de los elementos dibujados a fin de saber cuándo el usuario ha pulsado sobre uno de ellos.

### Dibujar ubicación de los contactos

Como ya se explicó en el apartado 4.2.1, el paquete `com.google.android.maps` es el que permite trabajar con los mapas y servicios ofrecidos desde Google Maps. En este paquete existe una clase, de nombre `Overlay`, que es la que permite dibujar elementos sobre el mapa.

Esta clase se diferencia notablemente de otras clases utilizadas por Android para dibujar elementos en pantalla, clases que se verán más adelante. La clase `Overlay` es específica para mapas y asocia a cualquier objeto dibujado los movimientos y cambios que se realicen sobre dichos mapas. Dicho de otro modo: lo habitual es que si el desarrollador desea situar elementos en un mapa, también desee que estos mismos elementos se desplacen coherentemente según el usuario desplace el mapa, o aparezcan o desaparezcan de la pantalla según el nivel de zoom que se haga sobre él. Es, por ejemplo, lo que debe ocurrir en *ContactMap* con los contactos y elementos derivados que se dibujan. Todos estos detalles de desplazamiento conjunto quedan cubiertos por la clase `Overlay`.

En esta clase se encuentra un método fundamental llamado `draw()`. Este método es constantemente invocado cada vez que ocurre algún cambio en el mapa; por ejemplo, cuando el usuario pulsa en él, cuando lo desplaza o cuando realiza cambios en el nivel de zoom. El método `draw()` debe contener aquellos elementos que se deseen dibujar de forma que, si lo elementos están asociados a coordenadas, solamente serán dibujados si dado el mapa mostrado en pantalla deben ser realmente visibles para el usuario.

La única forma de implementar a medida este método es extendiendo la clase `Overlay` mediante otra clase. En *ContactMap*, la clase `MapOverlay` extiende a dicha clase y dibuja, con su correspondiente método `draw()`, los contactos y demás elementos asociados.

A continuación se detalla, en el Código 22, el fragmento del método `draw()` de `MapOverlay` que dibuja en el mapa los contactos, representándolos mediante un punto de color y acompañándolos de su nombre respectivo.

```
public class MapOverlay extends Overlay{

@Override
public void draw(Canvas canvas, MapView mapView, boolean shadow) {

      super.draw(canvas, mapView, shadow);

      // Punto de color
      Paint paintPoint = new Paint();
      // (...)
```

```
        // Circunferencia que rodea a un punto
        Paint paintCurrent = new Paint();
        // (...)

        // Texto con el nombre
        Paint paintText = new Paint();
        paintText.setColor(Color.BLACK);
        paintText.setTextSize(18);
        paintText.setFakeBoldText(true);

        // Recorrer contactos y bibujar los puntos de color
        while (mFriendList.hasNext()){

              // Obtener localización
              GeoPoint g = mFriendList.getGeo();

              // Proyectar localización en coordenadas de píxeles
              Point point = new Point();
              mMapView.getProjection().toPixels(g, point);

              // Establecer color para el punto
              i++;
              paintPoint.setColor(mColorsList.elementAt(i));

              // Dibujar punto
              canvas.drawCircle(point.x, point.y, 6, paintPoint);

              // Dibujar circunferencia si es el contacto actual
              if (mFriendList.equals(mCurrent)){
                    paintCurrent.setColor(mColorsList.elementAt(i));
                    canvas.drawCircle
                          (point.x, point.y, 30, paintCurrent);
              }

              canvas.drawText
               (mFriendList.getName(),point.x+3, point.y-12, paintText);

        }

}

}
```

**Código 22. Dibujado de contactos en el mapa**

En este código se utilizan algunas clases del paquete `android.graphics`. Este paquete alberga clases de todo tipo que resultan muy útiles para configurar y dibujar elementos en pantalla, desde figuras geométricas a textos o efectos. En concreto, en *ContactMap* se utilizan las siguientes clases:

- `Canvas`: clase a través de la cual se dibujan todos los demás elementos.

- `Paint`: define las características de un dibujo, como el color, el tamaño, tipo de letra para el texto, ciertos efectos, etc.
- `Color`: gestiona la manipulación de colores.
- `Point`: representa un punto mediante dos coordenadas (x, y).
- `RectF`: crea una figura con forma de rectángulo, según las coordenadas facilitadas.

Cada vez que se dibuja un elemento a través de `Canvas`, este debe estar asociado a un objeto `Paint` que configura el estilo del dibujo. Por ello, al principio del método `draw()` se definen tres objetos `Paint` que se asociarán, respectivamente, al punto de color, a la circunferencia concéntrica y al nombre del texto.

En el Código 22 solamente se muestra la configuración del objeto `Paint` llamado `paintText`, que representa el estilo deseado para dibujar el nombre del contacto al lado de su punto de color. Como se puede apreciar, con método como `setColor()`, `setTextSize()`, o `setFakeBoldText()` se establece el color deseado, el tamaño de texto o la presencia de negrita. Los otros objetos `Paint` obtienen sus configuraciones con métodos similares.

A continuación se recorren uno a uno los contactos presentes en la lista. El objetivo en cada recorrido es sencillo: transformar sus coordenadas de latitud y longitud en un punto de coordenadas (x, y) de la pantalla, dibujar ese punto con un color y escribir el nombre del contacto cerca del mismo.

La clase `MapView` (que, como se recordará, representa la vista de un mapa) tiene un método muy útil que permite proyectar unas coordenadas geográficas en coordenadas de un plano de dos dimensiones (x, y): el método `getProjection()`. De esta forma se obtiene un sencillo objeto `Point` que representa en qué punto de la pantalla se ubica el contacto, dadas su latitud y longitud actuales.

Después de elegir el color para el contacto de la lista `mColors`, se dibuja un punto de 6 píxeles de radio en su ubicación geográfica actual, transformada ya en unas coordenadas (x, y) que están presentes en el objeto `Point`. La clase `Canvas` permite dibujar un círculo con el método `drawCircle()`, donde simplemente se debe indicar las coordenadas de su centro y el radio deseado.


**Figura 23. Mapa con los contactos dibujados**

El foco de la aplicación *ContactMap* está siempre sobre alguno de los contactos mostrados, haciendo a éste objeto de todas las acciones posibles asociadas, como mandar un SMS, realizar una llamada o escribir un correo electrónico. En caso de que el contacto recién pintado sea el contacto actual, según marca el objeto `mCurrent` (que es una instancia de la clase `Friend`), se dibuja entonces una circunferencia entorno al punto con un radio amplio de 30 píxeles y del mismo color que el propio punto; así se indica que dicho usuario tiene el foco de la aplicación.

Por último, se debe escribir el nombre del contacto. El método `drawText()` de la clase `Canvas` permite dibujar textos como si de cualquier otro elemento se tratara. En este caso, el texto se sitúa en el área superior derecha con respecto al centro del punto dibujado.

Como información al lector, se dirá que el otro elemento dibujado, el bocadillo con información, no representa mayor dificultad de la vista hasta ahora. Simplemente consiste en una combinación de figuras geométricas que forman la silueta de un bocadillo cuyo centro se sitúa en el contacto y en el que se escribe dicha información.

Más adelante se describe el proceso mediante el cual se controlan la aparición o no de este bocadillo informativo.

Una vez explicado el funcionamiento de `MapOverlay` y como a través de una clase derivada `Overlay` se pueden pintar elementos en el mapa, es necesario mencionar ahora cómo se vincula un determinado `Overlay` a un mapa para dibujar elementos en él.

En la clase `ContactMap` que, como se ha dicho anteriormente, lleva el mayor peso de la ejecución y coordina las demás clases, se vincula al mapa mostrado el `Overlay` representado por la clase `MapOverlay`. En el Código 23 se enseña este proceso:

```
public class ContactMap extends MapActivity {

private MapView mMapView = null;                // Vista del mapa
private MapController mMapControl = null;       // Controlador del mapa

@Override
public void onCreate(Bundle savedInstanceState) {

      // ...

      // Crear Overlay asociado al MapView
      MapOverlay overlay =
                        new MapOverlay(this, mMapView, mMapControl);

      // Añadir Overlay a la lista de Overlays del MapView
      mMapView.getOverlays().add(overlay);

      // ...

}
}
```

**Código 23. Vincular un MapView con un Overlay**

Después de crear un objeto `MapOverlay`, éste se asocia al mapa representado por `mMapView` utilizando los métodos `getOverlays().add()`. Se puede intuir que un mismo mapa permite tener varios objetos `Overlay` asociados, de forma que se pueden pintar elementos de forma independiente a través de diferentes clases que se implementen. En este caso, el mapa mostrado por *ContactMap* solamente tendrá un `Overlay` asociado, encargado de dibujar los contactos y sus elementos asociados.

### Controlar los elementos dibujados

La clase `Overlay` no sólo da acceso a dibujar elementos asociados a un determinado mapa, sino que también permite controlar los eventos relacionados con ellos como, por ejemplo, que sean pulsados por el usuario.

El método `onTouchEvent()` se llama cada vez que el usuario pulsa en la pantalla. Este método se utiliza en `MapOverlay` (que extiende `Overlay`) para conocer si el usuario ha pulsado sobre el contacto y si es, por ello, necesario cambiar el foco, centrar el mapa o pintar un bocadillo con la fecha de actualización del contacto.

Existe un atributo en `MapOverlay` de carácter global llamado `mBubble`, que no es más que un tipo *string* donde se almacena el número de teléfono del usuario que ha sido pulsado o, en su defecto, almacena una cadena vacía como indicador de que ningún usuario debe mostrar un bocadillo.

El siguiente código muestra la parte más importante del método `onTouchEvent()`:

```
public class MapOverlay extends Overlay{


// Número del contacto que ha de mostrar un bocadillo

private String mBubble = new String("");

@Override
public boolean onTouchEvent(MotionEvent e, MapView mapView) {

      // Recorrer contactos
      while (mFriendList.hasNext()){

            // Proyectar contacto en coordenadas de píxeles
            Point punto = new Point();
            mMapView.getProjection()
                              .toPixels(mFriendList.getGeo(),punto);

            // Crear un rectángulo que abarque el punto
            RectF rec =
            new RectF(punto.x -8, punto.y -8, punto.x +8, punto.y +8);

            // Se ha pulsado dentro del rectángulo
            if (rec.contains(e.getX(), e.getY())){

                  // Se activa el bocadillo
                  if (mBubble.equals(""))
                        mBubble= mFriendList.getNumber();
                  // Se desactiva el bocadillo
                  else mBubble="";
```

```
                    // Se centra el mapa en el contacto pulsado
                    mMapControl.animateTo(mFriendList.getGeo());
                    // Se actualiza el contacto actual
                    mCurrent=friend;

                    return true;
                }
            }

            return false;
        }
    }
```

**Código 24. Control de pulsaciones sobre un contacto**

La estrategia seguida es muy sencilla. Se recorren todos los contactos dibujados en el mapa y se crea una zona en forma de rectángulo que abarque cada contacto recorrido. Después, se comprueba si el punto tocado por el usuario se encuentra ubicado dentro de esta zona, en cuyo caso será necesario dibujar el bocadillo o cambiar el contacto con el foco.

En primer lugar, de nuevo se obtiene una proyección de las coordenadas de latitud y longitud del contacto en coordenadas (x, y) de la pantalla. A continuación, se crea un rectángulo, representado por la clase `RectF`, cuyo centro sea el propio contacto. El rectángulo crea un área mayor que el propio punto, de forma que no sólo se activa el bocadillo si se pulsa en el propio contacto, sino que también lo hace si se pulsa en una zona relativamente próxima, lo que repercute en la comodidad del usuario.

Es importante hacer notar que el rectángulo **no se dibuja**, sino que simplemente se crea. Es un objeto más situado en ciertas coordenadas del mapa, pero no ha sido dibujado en él puesto que no se ha utilizado ningún objeto `Canvas` para ello. Existe, pero el usuario ni lo sabe ni lo puede percibir.

El método `onTouchEvent()` facilita las coordenadas (x, y) del punto donde el usuario ha pulsado. Conociendo las coordenadas de los contactos y las coordenadas del punto pulsado, simplemente es necesario comparar ambos; salvo que, en realidad, no se compara con el punto exacto del contacto sino que se hace con un área completa entorno a él, representada en el rectángulo creado.

La clase `RectF` permite saber si un punto dado se encuentra dentro del área del rectángulo mediante el método nativo `contains()`. Así pues, en caso de que el usuario haya pulsado en esta área que representa el contacto, se almacena su número de teléfono en `mBubble`, se convierte en el contacto que posee el foco a partir de ese momento y se centra el mapa en él. La próxima vez que el método `draw()` sea llamado (cosa que ocurrirá a continuación), todos estos cambios serán visibles para el usuario.

### 4.2.4 Control del mapa: movimiento y zoom.

En este apartado se enseña como tomar el control de mapa en dos vertientes: el control del nivel de zoom, y el libre desplazamiento mediante el sistema de pinchar y arrastrar.

Android cuenta con un paquete muy completo para la composición de elementos de interfaz, también llamado *widgets*, como botones, imágenes, cajas de texto, etc. Este paquete, denominado `android.widget` contiene clases como las siguientes:

- `Button`: representa un botón que puede ser pulsado por el usuario.
- `CheckBox`: permite incluir elementos *checkbox* en formularios.
- `DigitalClock`: muestra un reloj digital. La clase `AnalogClock` hace lo propio, pero con un reloj analógico.
- `EditText`: clase que muestra una caja de texto donde el usuario puede escribir.
- `FrameLayout`: clase que representa un área de la pantalla donde colgar otros elementos visuales.
- `ImageButton`: representa un botón al que se le puede asociar una imagen.
- `MediaController`: clase que ofrece un conjunto de controles para elementos multimedia.
- `ProgressBar`: muestra una barra de progreso.
- `RadioButton`: permite introducir elementos radiobutton en formularios.
- `TextView`: clase que ofrece un área de texto no editable; por ejemplo, títulos para campos de formularios.
- `VideoView`: posibilita introducir elementos de video.
- `ZoomControls`: ofrece elementos de control de zoom.

Android permite asociar vistas predefinidas a los elementos mostrados en la pantalla, de forma que estos se agrupan siguiendo determinado patrones de diseño; estos patrones predefinidos representan los más comúnmente utilizados en ciertos tipos de aplicaciones. Por ejemplo, existen en este paquete `android.widget` diversas clases como `Gallery`, `GridView`, `LinearLayout`, `ListView` o `TableLayout`.

En *ContactMap*, la interfaz mostrada no incluye solamente el mapa con los contactos dibujados en él; también se deben incluir ciertos *widgets* a través de los cuales el usuario pueda realizar algunas de las acciones contempladas en la aplicación. Una de estas acciones es el control del nivel de zoom del mapa.

La clase `FrameLayout` del paquete `android.widget` puede utilizarse para construir interfaces de usuario en las cuales existen diversos elementos. Esta clase define un área, que puede ser toda la pantalla o una parte de ella, donde colgar los elementos que componen la interfaz. Cada elemento deseado se añade a `FrameLayout` pudiendo además especificar ciertas características del mismo, como su posición relativa o su comportamiento frente a otros elementos presentes en el mismo `FrameLayout` o frente a ciertos eventos, como pulsaciones del usuario. Utilizando la clase `FrameLayout` es como se ofrece al usuario de *ContactMap* el control del nivel de zoom del mapa.

El siguiente código muestra los pasos a seguir para poner a disposición del usuario estos controles.

```
public class ContactMap extends MapActivity {

private MapView mMapView = null;    // Vista del mapa

@Override
public void onCreate(Bundle savedInstanceState) {

      // ...

      // Crear FrameLayout sobre el que colgar los widgets
      FrameLayout frame = new FrameLayout(this);

      // Colgar el mapa en el FrameLayout
      frame.addView(mMapView);

      // Obtener controles del zoom
      View zoomMapControls = mMapView.getZoomControls();

      // Configurar ubicación del zoom en el FrameLayout
      FrameLayout.LayoutParams p = new FrameLayout.LayoutParams(
            LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT,
            Gravity.BOTTOM + Gravity.CENTER_HORIZONTAL);

      // Colgar los controles de zoom en el FrameLayout
      frame.addView(zoomMapControls, p);

      // Visualizar FrameLayout
      setContentView(frame);

      // ...

}
}
```

**Código 25. Control del nivel de zoom**

Tras crear un nuevo objeto `FrameLayout`, ya se le pueden asociar distintos elementos de vista. El primero de ellos en ser colgado en el nuevo `FrameLayout` es el propio mapa. Como no se indica ningún parámetro de configuración, el mapa ocupará por defecto todo el área de este `FrameLayout`, es decir, la pantalla completa.

A continuación, se crea un objeto `ZoomControls`, clase que también pertenece al paquete `android.widget`. Sin embargo, es necesario que este control del zoom esté asociado al mapa, ya que es el objeto que debe ser controlado. Para ello, se utiliza el

método `getZoomControls()` de la clase `MapView`, que directamente devuelve un control de zoom asociados al mismo mapa.

Para los controles de zoom, sí se desea crear una configuración que matice su comportamiento dentro del `FrameLayout`, por lo que se añadirán a este incluyendo un objeto `FrameLayoutParams` con determinados valores. Con la constante `LayoutParams.WRAP_CONTENT`, se especifica que el objeto ocupe tanto espacio como necesite, y con `Gravity.BOTTOM` y `Gravity.CENTER_HORIZONTAL` se establece que el objeto se sitúe en la parte inferior del objeto `FrameLayout` y centrado, respectivamente.

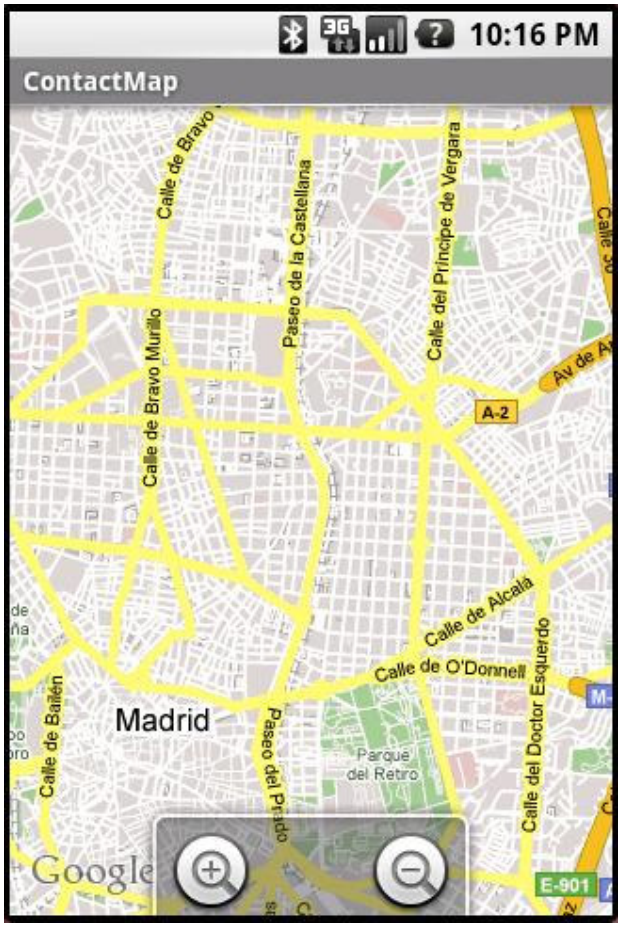


**Figura 24. Controles del nivel de zoom**

Por defecto, el control del nivel de zoom se hace visible al usuario cuando este pulsa en la pantalla, y se vuelven a ocultar tras unos pocos segundos de inactividad.

Para permitir al usuario que pueda desplazarse por el mapa libremente simplemente arrastrando el dedo por la pantalla, es tan simple como invocar el método

`setClickable()` de la clase `MapView` y establecer el valor `true`. De este modo, además se permite capturar los eventos relacionados con pulsaciones en la pantalla.

### 4.2.5 Cambiar al contacto anterior y siguiente

El contacto que mantiene el foco de la aplicación es aquel que es objeto de algunas de las acciones que *ContactMap* ofrece, como realizar una llamada, enviar un SMS o un correo electrónico. El contacto que tiene el foco está representado por un círculo que rodea al punto de color que lo representa en el mapa.

El usuario puede cambiar el foco pulsando sobre otro contacto, eligiéndolo de un listado de contactos o, tal y como se explica en este apartado, utilizando unos botones para desplazarse al contacto anterior o al siguiente, según su orden alfabético. Estos botones han de estar siempre presentes en la pantalla y ubicados en las esquinas inferiores de la interfaz principal, es decir, sobre el mapa con los contactos dibujados.

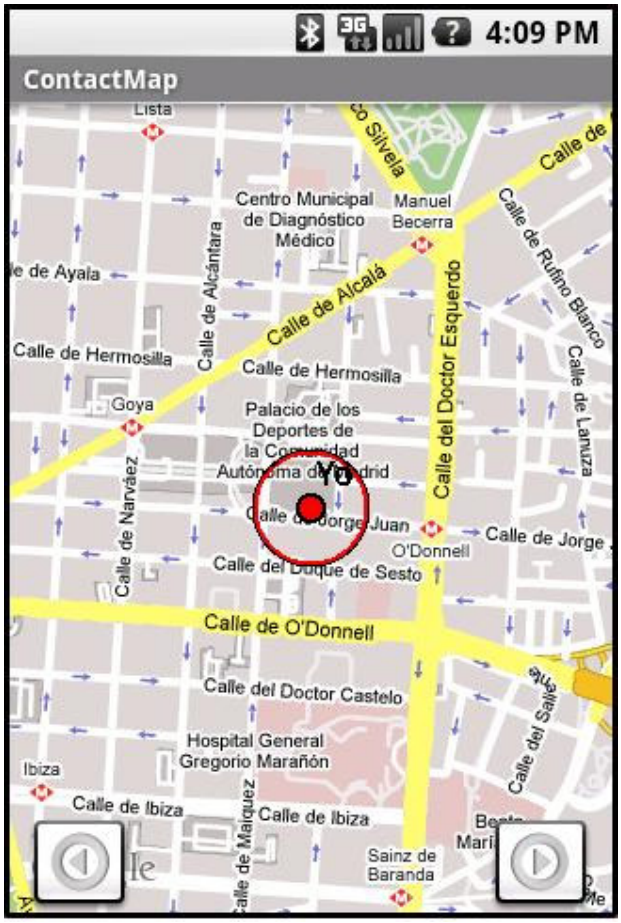


**Figura 25. Botones de anterior y siguiente**

Android permite asociar a una imagen cualquiera de las propiedades de un botón, entre las que se encuentra principalmente el que pueda ser pulsada por el usuario y asociar tal evento a una determinada acción. La clase que permite este comportamiento es `ImageButton`, del paquete `android.widget`.

A continuación se enseña al lector el fragmento del método `onCreate()` de la clase `ContactMap` que crea el botón que permite mover al siguiente contacto, siguiendo un orden alfabético.

```
public class ContactMap extends MapActivity {

@Override
public void onCreate(Bundle savedInstanceState) {

      // (...)

      // Crear botón Siguiente
      ImageButton btnNext = new ImageButton(this);

      // Asociarle imagen de la carpeta \res\drawable
      btnNext.setImageResource(R.drawable.siguiente);

      // Configurar ubicación del botón en el FrameLayout
      FrameLayout.LayoutParams sigLayout = new
            FrameLayout.LayoutParams(LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT, Gravity.BOTTOM + Gravity.RIGHT);

      // Añadir botón al FrameLayout
      frame.addView(btnNext,sigLayout);

      // Implementar método onClick() del botón
      btnNext.setOnClickListener(new View.OnClickListener() {
            public void onClick(View view) {
                  moveNext();
            }
      });

      // (...)

}
}
```

**Código 26. Mostrar botón de siguiente contacto**

En primer lugar, se instancia un objeto de la clase `ImageButton`, pasando únicamente como parámetro un objeto `Context` válido, que representa el contexto de la aplicación. La misma clase `ContactMap` sirve como parámetro de este tipo.

Para asociar una imagen a un objeto `ImageButton` no es necesario más que invocar el método `setImageResource()`. La imagen que se utiliza para representar este botón se encuentra ubicada en la carpeta de recursos "\res\drawable" bajo el nombre de "next.png".

Para configurar la ubicación de la imagen, se utiliza la clase ya anteriormente vista `FramLayout.LayouParams`. En este caso, se especifica que la imagen ocupe todo el espacio que necesite (es una imagen muy pequeña) y en la parte inferior derecha de la pantalla.

El lector probablemente recuerde la clase `FrameLayout` que permitía definir un área donde poder colgar elementos para componer la interfaz de usuario. En el apartado 4.2.4 se utilizaba dicha clase para incluir botones para el control del nivel de zoom. En esta ocasión, se utiliza el mismo objeto `FramLayout` para incluir, mediante su método nativo `addView()`, un nuevo elemento a la interfaz: el botón de siguiente.

Tras mostrar el botón en la interfaz de usuario, es necesario asociarle el comportamiento deseado. Utilizando un *listener* vinculado a dicho botón, se implementa el método `onClick()`, que provocará que se llama a `moveNext()` cada vez que el usuario pulse este botón. El método `moveNext()`, también definido en `ContactMap`, únicamente recorre la lista de contactos y establece el foco en el contacto que es siguiente al actual.

El proceso aquí descrito es exactamente igual para el botón que permite al usuario desplazarse al contacto anterior.

### 4.2.6 Control de la señal GPS

*ContactMap* utiliza la señal GPS para conocer la ubicación actual del usuario, pudiendo así tanto dibujarla en el mapa mostrado como comunicársela a los demás usuarios a través de su actualización en el servidor.

#### Conocer la propia ubicación

Para poder acceder y controlar a la señal GPS se utilizan las clases presente en el paquete `android.location`. En general, este paquete incluye clases que permiten manejar diferentes dispositivos de localización. Algunos de los elementos presentes en este paquete son:

- `Location`: clase que representa una localización geográfica.
- `LocationManager`: clase para gestionar el dispositivo de localización.
- `Geocoder`: clase que permite traducir entre direcciones físicas y coordenadas en latitud y longitud, y viceversa.
- `LocationListener`: interfaz que permite implementar una clase que captura los eventos asociados al dispositivo de localización.

Las clases básicas para obtener la señal de GPS son `LocationManager` para controlar el dispositivo, `LocationListener` para escuchar sus cambios y `Location` para guardar la información de localización. En la clase `ContactMap` está declarada una variable global para controlar el dispositivo, llamada `mLocation`, y se define un subclase llamada `MyLocationListener` que atiende los posibles eventos asociados al GPS.

En primer lugar, es necesario acceder al dispositivo de localización, el GPS en este caso. Para acceder a este o a cualquier otro servicio integrado en el dispositivo móvil, se utiliza el método `getSystemService()` de la clase *`Activity`* y sus clases derivadas, como es `ContactMap`. Para referenciar al GPS es necesario pasar la constante `Context.LOCATION_SERVICE`. De esta forma, en `mLocation` se tendrá un controlador válido para el dispositivo GPS.

Antes de poder obtener la posición actual a través de `mLocation`, Android precisa que se defina una clase que permita gestionar sus posibles eventos asociados al GPS, como que la posición cambie (exista un desplazamiento del usuario) o que el dispositivo de localización se active o desactive. Para ello, ha de definirse una clase que implemente la interfaz `LocationListener`. Esta clase se implementa dentro de `ContactMap`, y recibe el nombre de `MyLocationListener`. La varible global `mListener` es una instancia de esta clase. Así pues, una vez definida la interfaz se asocia dicha clase al controlador del GPS con el método `requestLocationUpdates()` de `mLocation`..

Además, es necesario otorgar a la aplicación permiso correspondiente para acceder al dispositivo de localización mediante la declaración en el fichero "AndroidManifest.xml" de la etiqueta correspondiente:

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

**Código 27. Declaración en el manifiesto del permiso de acceso al GPS**

En este punto ya es posible acceder a la localización del usuario usando el método `getLastKnownLocation()` del objeto `mLocation`, que es el que tiene el control del GPS. La localización vendrá dada en un objeto `Location`, y de ella se puede obtener, entre otras cosas, la latitud y longitud con los métodos `getLatitude()` y `getLongitude()`, respectivamente.

A continuación se muestra el código correspondiente. De nuevo, se declaran en el método `onCreate()` de la clase `ContactMap`, para poder realizarse nada más arrancar la aplicación. El fragmento de código mostrado se centra únicamente en lo explicado en líneas anteriores, omitiendo lo demás:

```
public class ContactMap extends MapActivity {

// Controlador del localizador GPS
private LocationManager mLocation = null;
// Listener para el localizador (captura eventos)
private MyLocationListener mListener = null;

/** Called when the Activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      // (...)

      // Obtener controlador de GPS
      mLocation = (LocationManager)
                  getSystemService(Context.LOCATION_SERVICE);
      // Asociar un listener
      mListener = new MyLocationListener();
      mLocation.requestLocationUpdates(
                                    LocationManager.GPS_PROVIDER,
                                    0,
                                    0,
                                    mListener);

      // Conocer ubicación actual
      Location miLoc =
      mLocation.getLastKnownLocation(LocationManager.GPS_PROVIDER);

      // (...)

}

}
```

**Código 28. Acceso al dispositivo GPS y a la posición actual**

### Capturar eventos del GPS

La implementación de la clase `MyLocationListener`, encargada de escuchar los eventos del GPS, es muy simple. Cada vez que se detecta que la posición del GPS cambia, se invoca el método `onLocationChanged()` de esta clase. La única acción que es necesario hacer cuando cambie la posición del usuario es actualizarla para poder mostrarla correctamente en el mapa e informar al servidor en la próxima conexión. Como se ha explicado en un apartado anterior (ver 4.2.2) `ContactMap` mantiene en todo momento una lista de los contactos que se pueden mostrar en el mapa llamada `mFriendList`, donde además se incluye al propio usuario. Es en esta lista donde se busca y se actualiza la nueva localización del usuario.

El Código 29 expone la implementación básica de la clase `MyLocationListener`. Únicamente se completa el método `onLocationChanged()` que es el necesario para actualizar la ubicación del usuario.

```
private class MyLocationListener implements LocationListener{

/**
 * Llamado cuando cambia la propia ubicación.
 * Actualiza los datos de localización.
 */
@Override
public void onLocationChanged(Location loc) {

      // Componer GeoPoint con los datos de localización
      GeoPoint myGeo = new GeoPoint(
                                    location.getLatitude()*1E6,
                                    location.getLongitude()*1E6);

      // Buscar a usuario en la lista mFriendList
      while (mFriendList.getName.equals("Me")){
            mFriendList.next();
      }

      me=mFriendList;
      // Actualizar datos de localización y fecha
      me.setGeo(myGeo);
      me.setDate(date);
}
}

@Override
public void onProviderDisabled(String provider) {
      // Llamado cuando el usuario desactive el localizador
}

@Override
public void onProviderEnabled(String provider) {
      // Llamado cuando el usuario active el localizador
}

@Override
public void onStatusChanged(String provider, int status, Bundle
extras) {
      // Llamado cuando cambie el estado de localizador

}
}
```

**Código 29. Implementación de la clase MyLocationListener**

### Ventana de alerta al usuario

Pueden existir casos donde el dispositivo móvil no disponga de localizador GPS, o donde éste se desactive por el usuario o no pueda facilitar una señal. En ese caso, `ContactMap` no podrá mostrar la ubicación del usuario ni notificársela al servidor, por lo que no será conocida por el resto de usuarios.

Para conocer el estado del GPS, se utiliza el método `isProviderEnabled()` del controlador `mLocation`. Este simplemente devuelve un valor booleano donde indica si el GPS está activado o no. En caso de estar activado, se utiliza el GPS sin mayor problema. Si no, la aplicación *ContactMap* lanza un aviso al usuario.

El siguiente código muestra cómo se realizan estos dos casos en la aplicación.

```
public class ContactMap extends MapActivity {

/** Called when the Activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {

      // ...

      // Comprobar disponibilidad del GPS
      boolean gpsEnabled =
      mLocation.isProviderEnabled(LocationManager.GPS_PROVIDER);

      // GPS disponible
      if (gpsEnabled==true){

            // Obtener localización del usuario

      }else{
            // GPS no disponible, lanzar aviso

            AlertDialog.Builder alertNoGPS=
                                    new AlertDialog.Builder(this);
            alertNoGPS.setTitle(R.string.noGPStitle);
            alertNoGPS.setMessage(R.string.noGPSmessage);
            alertNoGPS.setNeutralButton(R.string.neutralButton,null);
            alertNoGPS.show();
      }

      // ...

}
}
```

**Código 30. Comprobación del estado del GPS**

La clase `AlertDialog`, del paquete `android.app`, permite lanzar ventanas de aviso al usuario asociando a esta texto, botones, o incluso vistas más completas. En el Código 30 se aprecia como se incluye un título con `setTitle()`, un mensaje con `setMessage()` y un botón sin acción asociado con `setNeutralButton()`.

Todos estos mensajes de texto no se incluyen de forma directa en el código, sino que se adjuntan como recurso externo. En concreto, han sido situados en la carpeta "\res\values", dentro de un fichero que debe llevar por nombre "strings.xml". De esta forma, Android facilita la inclusión de diferentes versiones de texto, por ejemplo, para cambiarlos de idioma. El texto declarado en "strings.xml" es el siguiente:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<resources>
      (...)
      <string name="noGPStitle">Sin localización</string>
      <string name="noGPSmessage">No hay dispositivo de localización
      habilitado. No se mostrará la propia ubicación</string>
      <string name="neutralButton">Aceptar</string>
      (...)
</resources>
```

**Código 31. Declaración de recursos de texto**

No debe olvidarse referencia el tipo de codificación "iso-8859-1" para poder incluir caracteres con tildes y la "ñ". En caso contrario, Android dará un mensaje de error de difícil interpretación.

### Establecer ubicación propia en el emulador

Si se está trabajando sobre el emulador de Android, como es el caso, no se dispondrá de información de localización a través de ningún dispositivo GPS. Cuando se realicen las llamadas correspondientes, la información que se devuelve es la que esté almacenada en el emulador.

Para fijar en el emulador unas coordenadas geográficas, deben realizarse los siguientes pasos:

1. Abrir una línea de comandos con *Inicio >Ejecutar >cmd*
2. Lanzar una conexión telner al emulador, mediante la instrucción:

```
> telnet localhost 5554
```

3. Introducir los datos de coordendas con la siguiente orden, sustituyendo LONGITUD y LATITUD por los valores deseados:

```
> geo fix LONGITUD LATITUD
```

### 4.2.7 Control de la señal Wi-Fi

*ContactMap* conecta con el servidor de forma periódica para actualizar la localización del usuario y para conocer las localizaciones de sus contactos. Esta comunicación se realiza a través de Internet con conexiones HTTP.

El paquete `android.net.wifi` provee a Android de diversas clases para gestionar el dispositivo Wi-Fi del aparato móvil. En concreto, se utilizará la clase `WifiManager` para controlar este elemento de conexión.

Al igual que con el GPS, al dispositivo de Wi-Fi se accede utilizando el método `getSystemService()` de la clase `ContactMap`, que da acceso a distintos servicios integrados en el dispositivo móvil. En esta ocasión, la llamada debe realizarse utilizando la constante `Context.WIFI_SERVICE`. De esta forma, se adquiere un objeto `WifiManager` válido con el que controlar la Wi-Fi.

Para poder utilizar sin problemas el dispositivo de Wi-Fi, Android exige que se autorice expresamente a la aplicación a usar dicho dispositivo. Esta autorización se manifiesta a través del permiso correspondiente en el fichero "AndroidManifest.xml":

```
<uses-permission android:name=
                              "android.permission.ACCESS_WIFI_STATE" />
```

**Código 32. Declaración en el manifiesto del permiso de acceso al dispositivo Wi-Fi**

La ausencia de conectividad a través de Wi-Fi provoca que *ContactMap* no pueda conectar con el servidor y, por tanto, tampoco actualizar ni su propia ubicación ni la de sus contactos. Sin embargo, en ese caso *ContactMap* muestra la última ubicación conocida de cada contacto **gracias al uso de la base de datos** SQLite, donde se van guardando las localizaciones conocidas. La utilización detallada de esta base de datos se expone en el apartado 4.2.12.

El siguiente código muestra el acceso a la Wi-Fi del dispositivo móvil, así como la comprobación de su estado.

```
public class ContactMap extends MapActivity {

/** Called when the Activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {

      // ...

      // Obtener controlador de Wi-Fi
      WifiManager wifi = (WifiManager)
                        getSystemService(Context.WIFI_SERVICE);
      // Comprobar disponibilidad de Wi-Fi
      boolean wifiEnabled = wifi.isWifiEnabled();

      // Wi-Fi disponible
      if (wifiEnabled==true){

            // Lanzar la clase Update

      }else{
            // Wi-Fi no disponible, lanzar aviso

            AlertDialog.Builder alertNoWiFi=new
            AlertDialog.Builder(this);
            AlertDialog.setTitle(R.string.noWIFItitle);
            AlertDialog.setMessage(R.string.noWIFImessage);
            AlertDialog.setNeutralButton(R.string.neutralButton,null);
            alertNoWiFi.show();

            updateFriendsFromDataBase();

            }
}
}
```

**Código 33. Acceso y comprobación del dispositivo de Wi-Fi**

Para conocer el estado de la conexión Wi-Fi se utiliza el método `isWifiEnabled()` de la clase `WifiManager`, a la que pertenece el objeto instanciado `wifi`. Si está disponible, se lanza el servicio `Update` sin más (ver más adelante, apartado 4.2.18). Si no, se lanza una nueva ventana de aviso al usuario, y se actualiza la lista `mFriendList` desde la base de datos local SQLite en lugar desde el servidor remoto, llamando al método `updateFriendsFromDataBase()`.

Nótese como, de nuevo, se han utilizado en la ventana de alerta al usuario cadenas de texto declaradas en el archivo "strings.xml", que se encuentra dentro de la carpeta de recursos "\res\values".


**Figura 26. Mensaje cuando no hay señal Wi-Fi disponible**

El SDK 1.0 de Android permite conocer qué tipo de conexiones existen en el dispositivo móvil (Wi-Fi, GPRS, UMTS, Bluetooth, etc.), así como controlar su estado y el componente hardware correspondiente; sin embargo, esta versión del SDK no ofrece la posibilidad de elegir qué conexión se desea utilizar para el envío y la recepción de datos: por defecto, Android siempre intenta utilizar la conexión Wi-Fi, y en caso de no estar disponible utiliza las demás, consideradas de menor prioridad. Debido a este comportamiento tan peculiar, en la aplicación *ContactMap* se ha optado por realizar las conexiones con el servidor **únicamente cuando la conexión Wi-Fi esté habilitada**, descartando así realizar conexiones bajo otros protocolos que pueden llevar algún coste asociado para el usuario.

Además, en el caso de la conexión mediante Bluetooth, el SDK 1.0 limita sus funciones a la conexión de periféricos inalámbricos como auriculares, altavoces, manos libres, o similares. Es decir, actualmente el intercambio de datos mediante Bluetooth no está soportado por Android. Esta más que sorprendente decisión es argumentada por Google exponiendo problemas de falta de tiempo para depurar y certificar las API relacionadas con este estándar de comunicación, prometiendo además su pronta incorporación en siguientes versiones estables del SDK [39].

### 4.2.8 Conexiones HTTP

Los intercambios de información en formato XML entre la aplicación *ContactMap* y el servidor se realizan mediante el protocolo HTTP. En las peticiones se utiliza el método POST de dicho protocolo, encapsulando el documento XML correspondiente dentro de un campo de nombre "xml". En las respuestas, el contenido XML viaja en el campo de datos.

Para estos flujos de información se utilizan las clases `Update` y `HTTPConnection`. En concreto, es esta última la encargada de las labores de más bajo nivel a la hora de establecer una conexión HTTP.

Android cuenta con numerosos paquetes y clases dedicados en exclusiva a los diferentes aspectos de las comunicaciones. Por ejemplo, de Java hereda, entre otros, `javax.net` o `java.net`. Dentro de sus paquetes exclusivos como plataforma, incluye una amplísima familia de paquetes bajo el nombre raíz `android.net.*` y sobre todo `org.apache.http.*`, de la que se hará uso en esta aplicación.

La clase `HTTPConnection` define tres variables globales de ámbito privado. Por un lado, esta `mServer` que indica la dirección a la que se deben dirigir las conexiones. Por otro, están `mDataOut` y `mDataIn` que representan, respectivamente, los datos que se envían y los datos recibidos. El único método relevante de esta clase es `connect()`, que utilizando los valores dados a estas variables establece una conexión por la que enviar una petición (`mDataIn`) y recibir su respuesta (`mDataOut`).

Antes de iniciar la conexión, se construye la petición HTTP. La clase `BasicNameValuePairs` permite crear pares del tipo variable-valor que se enviarán junto a una petición POST. Así pues, se define una única variable de nombre "xml" que contendrá como valor el existente en `mDataOut`, es decir, el documento XML con la petición a enviar al servidor. Este contenido ha de recibir el formato adecuado de codificación de caracteres para poder viajar dentro del protocolo HTTP. El formato correcto se le aplica con la clase `UrlEncodedFormEntity`.

Una vez se tiene preparados los datos de envío, se define la petición POST completa usando la clase `HttpPost`. En su constructor, se especifica con `mServer` la dirección completa del extremo destino, y con los método `setHeader()` y `setEntity()` se asocian las cabeceras requeridas y los datos para enviar.

A continuación se crea el cliente HTTP con la clase `DefaultHtttpClient`. Ésta cuenta con el método `execute()`, que precisa como parámetros la petición a enviar y se encarga de ejecutar la conexión. Para recibir la correspondiente respuesta, se define un objeto `HttpResponse` que recibe el resultado de `execute()`.

Para finalizar, la respuesta se procesa mediante las clases `InputStreamReader` y `BufferedReader` para poder ser finalmente almacenada en `mDataOut`, desde donde será derivada a la clase `XMLExchange` y `MyXMLHandler` para su procesado.

En el Código 34 se muestran los pasos para establecer y utilizar una conexión HTTP exitosamente.

```
public class HTTPConnection {

private String mDataIn;       // datos recibidos
private String mDataOut;      // datos para enviar
private String mServer;       // servidor

public void connect(){

      // Preparar los datos que se enviarán
      List<BasicNameValuePair> pairs = new
                                    ArrayList<BasicNameValuePair>();
      pairs.add(new BasicNameValuePair("xml", mDataOut));
      UrlEncodedFormEntity entity = new UrlEncodedFormEntity(pairs);

      // Cabeceras y datos
      HttpPost httppost = new HttpPost(mServer);
      httppost.setHeader(
            "Content-Type", "application/x-www-form- urlencoded");
      InetAddress localhost = InetAddress.getLocalHost();
      httpPost.setHeader("Host", localhost.getHostAddress());
      httppost.setEntity(entity);
      // Cliente HTTP
      DefaultHttpClient httpclient = new DefaultHttpClient();

      // Conectar
      HttpResponse response = httpclient.execute(httppost);

      // Crea el canal de entrada
      BufferedReader entrada = new BufferedReader(
            new InputStreamReader(response.getEntity().getContent())
      );

      String linea;

      // Leer el buffer de entrada
      while( (linea = entrada.readLine()) != null ) {
            mDataIn=mDataIn.concat(linea);
      }

}
}
```

**Código 34. Conexión HTTP para enviar y recibir datos.**

Para poder realizar una conexión es necesario tener declarado en el manifiesto el permiso que da acceso a Internet `android.permission.INTERNET`. Este permiso, como ser recordará, ya fue declarado con anterioridad para poder hacer uso de los servicios de Google Maps.

### 4.2.9 Procesado de peticiones y respuestas en XML

Todos los intercambios que existen entre la aplicación *ContactMap* y el servidor se realizan utilizando documentos XML, tal y como se detalló en el apartado anterior 4.1.3. Las tareas relacionadas con la escritura y lectura de documentos XML para los intercambios de datos se realizan en las clases `XMLExchange` y `MyXMLHandler`.

La clase `XMLExchange` es la encargada de escribir las peticiones que se enviarán al servidor, así como de procesar las respuestas recibidas desde este. Define dos únicos métodos:

- `writeXMLUpdate()`: compone las peticiones hacia el servidor. Utiliza tanto la información obtenida del GPS y del dispositivo móvil para escribir los datos propio usuario, como de los contactos almacenados en el mismo dispositivo móvil para incluir sus números de teléfono.

- `readXMLUpdate()`: procesa las respuesta enviadas por el servidor, utilizando la clase `MyXMLHandler` en dicha tarea.

Para poder procesar con mayor facilidad y rapidez un documento XML, existe en Java una popular librería con tal fin llamada SAX. Esta permite recorrer los elementos y sus contenidos de forma automática. Con tal fin, la clase `MyXMLHandler` extiende a la clase `DefaultHandler`. De este modo, consigue parsear el documento recibido, pudiendo asociar una determinada acción a la lectura de cada uno de los elementos presentes en el árbol XML.

La respuesta consta, tal y como se describe en el DTD adjunto en el apartado 4.1.3, de elementos `<contact>` que albergan los datos del contacto. Cada vez que `MyXMLHandler` encuentra uno de estos elementos, representantes de un determinado contacto, actualiza los datos recién recibidos de dicho contacto en la lista `mFriendList` de la clase `ContactMap`.

```
public class XMLExchange {

public void readXMLUpdate (String XMLdata, ArrayList<Friend>
friendList, ContentResolver content){

      // Obtener un SAXParser desde SAXPArserFactory
      SAXParserFactory spf = SAXParserFactory.newInstance();
      SAXParser sp = spf.newSAXParser();

      // Obtener un XMLReader desde el SAXParser
      XMLReader xr = sp.getXMLReader();
      // Control de los eventos del XMLReader
      MyXMLHandler myHandler = new MyXMLHandler(friendList,content);
      xr.setContentHandler(myHandler);
```

```
        // Transformar los datos en un InputSource
        StringReader reader = new StringReader(XMLdata);
        InputSource source = new InputSource(reader);

        // Parsear los datos en XML
        // Se irán añadiendo a friendList
        xr.parse(source);

}
}
```

**Código 35. Procesado de la respuesta XML**

### 4.2.10 Acceso a los contactos del dispositivo móvil

Cuando *ContactMap* solicita al servidor la localización de sus contactos, utiliza el número de teléfono como clave identificativa. Así mismo, la aplicación escribe el nombre de cada contacto al lado de su ubicación en el mapa, y utiliza tanto el teléfono como la dirección de correo electrónico para enviar, respectivamente, un mensaje de texto o un correo. Toda esta información se obtiene de los contactos almacenados en el dispositivo móvil. Las tareas relacionadas con la manipulación de los contactos se lleva a cabo a través de la clase `Contacts`.

Para acceder a la información de contactos y, en general, acceder a diferentes tipos de contenedores de información, se necesitan tres elementos:

- Un objeto a través del cual acceder al contenedor de información deseado. En este caso, se utiliza la clase `ContentResolver`, del paquete `android.content`, que permite entre otras cosas lanzar consultas sobre un determinado recurso.
- Un objeto que recoja el resultado de la consulta. La clase `Cursor`, del paquete `android.database`, permite almacenar y recorrer el resultado obtenido al realizar una consulta a un contenedor de información.
- Una referencia que indique a qué contenedor concreto se desea acceder. En Android, los contenedores de información suelen estar referenciados por una URI que los identifica y que permite su acceso concreto. En el caso que nos ocupa, se utiliza la clase `Uri` del paquete `android.net.Uri`.

La clase `ContentResolver` utilizada para tener acceso a los contactos pertenece a uno de los paquetes más importantes de Android, el paquete `android.content`. Contiene clases que permiten acceder y publicar diferentes tipos de contenidos. Algunas de sus clases más relevantes son las siguientes:

- `ServiceConnection`: permite monitorizar el estado de un componente *Service*.
- `BroadcastReceiver`: uno de los componentes básicos de una aplicación Android, que permite a una aplicación escuchar y atender *Intents*.

- `ContentProvider`: otro de los componentes básicos de Android. Facilita a las aplicaciones publicar sus contenidos y hacerlos accesibles a otras aplicaciones del sistema.
- `ContentResolver`: ya mencionado, posibilita el acceso a modelos de contenidos.
- `Context`: clase que representa el ámbito de ejecución de una aplicación.
- `Intent`: clase de vital importancia en Android, que representa una acción y sus datos asociados. Recuérdese que, mediante un *Intent*, se lanza una solicitud para que determinada acción sea llevada a cabo por la aplicación más adecuada entre todas las disponibles en el sistema (ver más adelante en este mismo capítulo).

En la clase `Contacts` se define un total de tres métodos de acceso a la información de contactos. El método `getContacts()` devuelve un objeto `Cursor` que contiene todos los contactos presentes en el dispositivo móvil. Los otros dos métodos son de carácter más concreto: `getContactNameByNumber()`, que devuelve el nombre de un contacto según su número, y `getContactEmailByNumber()`, que obtiene el correo electrónico de un contacto también según su número de teléfono.

Los métodos mencionados de la clase `Contacts` acceden a la información de contactos de forma muy similar. Por ello, se explicará el más general de ellos, `getContacts()`. El Código 36 muestra su código fuente:

```
public class Contacts{
public Cursor getContacts(ContentResolver content){

      // Columnas que se desean obtener: nombre, número e ID
      String[] projection = new String[] {
                                    Contacts.People.NAME,
                                    Contacts.People.NUMBER,
                                    Contacts.People._ID};

      // Establecer URI para acceder a los contactos
      Uri contacts = Contacts.People.CONTENT_URI;

      // Lanzar consulta
      Cursor c = content.query(contacts,
                        projection,
                        null,
                        null,
                        Contacts.People.NAME + " ASC");

      c.moveToFirst();
      return c;
}
}
```

**Código 36. Acceso a los contactos del dispositivo móvil**

En primer lugar se construye un array de tipo *string* donde se especifican las columnas que se desean obtener en la consulta. La clase `Contacts.People` representa una de las

tablas que contiene información sobre los contactos, y forma parte de un interesante paquete llamado `android.providers`. Mediante este paquete se pueden referenciar las tablas y columnas de todos los contenedores de datos facilitados por Android, como pueden ser contactos, llamadas, fotografías, audios o vídeos, entre otros.

Para esta consulta se requieren tres columnas: el nombre del contacto, `Contacts.People.NAME`, su número de teléfono, `Contacts.People.NUMBER`, y su identificador único dentro del conjunto de contactos, `Contacts.People._ID`. Este último campo permite acceder a otras tablas donde el contacto está referenciado usando el identificador como clave ajena.

De momento, ya se ha indicado qué columnas interesan, pero no la tabla que debe ser consultada en busca de dichas columnas. El siguiente paso necesario es poder referenciar en la consulta a qué contenedor exacto vamos a consultar. Cada contenedor de datos de Android suele tener asociada una URI única que lo identifica. La constante `Contacts.People.CONTENT_URI` hace referencia a la URI que interesa para esta consulta, por lo que es almacenada en un objeto `Uri`.

Solamente resta lanzar la consulta al contenedor de contactos. El objeto `ContentResolver` nos permite utilizar el método `query()` donde, parámetro a parámetro, se puede construir una completa sentencia SELECT como se define en SQL estándar. Los parámetros del método `query()` son:

- la tabla que se desea consultar, identificada por un objeto `Uri`.
- las columnas que se quieren obtener, presentes en el array previamente construido.
- los filtros de selección deseados o cláusula WHERE; en este caso, ninguna.
- los valores utilizados en la filtración o cláusula WHERE; igualmente, ninguno en este caso.
- el orden del conjunto resultado: por nombre de contacto y en orden ascendente.

El resultado se aloja en un objeto `Cursor`, que permite recorrer y leer las filas obtenidas en la consulta. Para llevar a cabo este control del resultado, la clase `Cursor` incluye, entre otros, los siguientes métodos:

- `getCount()`: devuelve el número total de filas.
- `getPosition()`: devuelve la posición actual del cursor.
- `getString()`: devuelve una cadena con el valor de la columna indicada.
- `moveToNext()`: mueve el cursor a la siguiente fila.
- `moveToFirst()`: mueve el cursor a la primera fila.

El acceso a la información de contactos y, en general, a cualquier contenedor de información de Android requiere de los permisos necesarios expresados en el manifiesto. Así pues, en el fichero “AndroiManifest.xml” de `ContactMap` debe figurar una línea como la siguiente:

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

**Código 37. Declaración en el manifiesto del permiso de acceso a los contactos.**

### 4.2.11 Acceso a información sobre el dispositivo móvil

`ContactMap` no sólo obtiene información de localización de los contactos en el servidor, sino que además actualiza su propia información de localización para que esté disponible para los demás usuarios. Para ello, es necesario que *ContactMap* conozca el propio número de teléfono del usuario, ya que es la clave a través de la cual se identifica en el servidor.

La mejor forma de proporcionar el número de teléfono es preguntándoselo al propio dispositivo móvil. La clase `TelephonyManager`, incluida en el paquete `android.telephony`, permite esta consulta. Dicho paquete ofrece un pequeño conjunto de clases que permiten acceder a diferentes datos acerca del dispositivo móvil sobre el que Android se ejecuta.

El siguiente fragmento de código muestra parte del método `writeXMLUpdate()`, de la clase `XMLExchange`. Este método escribe una petición para el servidor, donde se incluyen los datos de localización del usuario, precedidos por su número de teléfono.

```
public class XMLExchange {

public String writeXMLUpdate(ContentResolver content, Context context,
Location location){

      StringBuffer XMLdata= new StringBuffer("");

      // Obtener controlador del dispositivo móvil
      TelephonyManager miTel = (TelephonyManager)
                        getSystemService(Context.TELEPHONY_SERVICE);
      // ...
      // Datos del propio usuario
      XMLdata.append("<me>");
      // Número de teléfono
      XMLdata.append("<number>");
      XMLdata.append(myTelephone.getLine1Number());
      XMLdata.append(("<number>");
      // ...
      XMLdata.append("</me>");
      // ...
}
}
```

**Código 38. Acceso al número de teléfono del dispositivo móvil**

El objeto `TelephonyManager` se obtiene utilizando el método `getSystemService()` que, como ya se vio con el GPS o la Wi-Fi, provee de los diferentes servicios integrados en el dispositivo móvil. Aquí se especifica la constante `Context.TELEPHONY_SERVICE` de la clase `Context`, que representa el contexto de la aplicación.

El método que proporciona el número de teléfono es `getLine1Number()`, pero la clase dispone de otros útiles métodos como `getCallState()`, que informa del estado de la llamada en curso, `getCellLocation()`, que informa de la celda GSM en la que se encuentra el dispositivo móvil, o `getNetworkOperatorName()`, que devuelve el nombre del operador de telefonía utilizado.

Como cabía esperar, el acceso a este tipo de información implica la declaración del siguiente permiso en el manifiesto de *ContactMap*:

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

**Código 39. Declaración en el manifiesto del permiso de acceso al dispositivo**

### 4.2.12 Uso de la base de datos SQLite

La información de localización de los contactos se actualiza de forma periódica mediante conexiones con el servidor. Sin embargo, en caso de no tener una señal Wi-Fi disponible *ContactMap* utiliza la información almacenada en la base de datos SQLite de Android para mostrar al menos las últimas ubicaciones conocidas.

Toda la gestión relacionada con la base de datos se lleva a cabo en la clase `FriendDataBase`. En ella están implementados dos métodos importantes: `getFriends()` que devuelve un objeto `Cursor` con todos los contactos almacenados en la base de datos, e `insertFriend()` que actualiza los datos de un contacto y en caso de no existir lo inserta. El manejo de la base de datos SQLite es muy similar al que se hace cuando se accede, por ejemplo, a la información de los contactos almacenados en el propio dispositivo móvil, ya explicado en el apartado 4.2.10.

En la clase `FriendDataBase` se declaran dos variables globales de ámbito privado. La primera, `mDataBase`, representa un objeto de la clase `SQLiteDatabase`. Esta clase forma parte del paquete `android.database.sqlite`, que incluye otras muchas clases e interfaces para el manejo de SQLite. La variable `mDataBase` representa un controlador de la base de datos y proporciona métodos para la creación, consulta o actualización de la misma. La otra variable global, llamada `mTable`, contiene la sentencia SQL necesaria para crear la base de datos en caso de ser la primera ejecución de *ContactMap*.

### Crear la base de datos

Como ya se ha mencionado, la variable `mTable` contiene la instrucción en SQL necesaria para crear la base de datos. Esta implica una única tabla donde se almacena número de teléfono, latitud, longitud y fecha de actualización de cada contacto para el que se ha obtenido alguna vez información desde el servidor. Se puede comprobar que su estructura, mostrada a continuación, es igual a la base de datos en MySQL residente en el servidor (ver apartado 4.1.4):

```
public class FriendDataBase {

      // Sentencia de creación de la BD
      private String mTable =
                  "CREATE TABLE IF NOT EXISTS friend " +
                  "(number INTEGER PRIMARY KEY, " +
                  "latitude REAL NOT NULL, " +
                  "longitude REAL NOT NULL, " +
                  "date TEXT NOT NULL)";
}
```

**Código 40. Script SQL de la base de datos SQLite de Android**

Cuando se llama al constructor de la clase, lo primero que se hace es abrir la base de datos, de nombre "ContactMapBD", utilizando para ello el contexto de la aplicación y el método `openOrCreateDatabase()`. Como es de suponer, en caso de no existir la base de datos con el nombre indicado, dicho método la crea.

A continuación, ya se dispone en la variable `mDataBase` de una controlador con el que manejar la base de datos. El siguiente paso es ejecutar el script SQL contenido en la variable `mTable`. Así, si la base de datos ha tenido que ser creada, se creará también la tabla necesaria. En caso de existir, la sentencia SQL no produce ningún efecto. Para lanzar la sentencia se utiliza el método `execSQL()` de la clase `SQLiteDatabase`.

En el Código 41. Creación de la base de datos SQLite.Código 41 se expone el constructor de `FriendDataBase`:

```
public class FriendDataBase {

public FriendDataBase(Context ctx){

      //apertura o creación de la base de datos
      this.mDataBase =
            ctx.openOrCreateDatabase("ContactMapBD",0,null);
```

```
      //creación de la tabla
      this.mDataBase.execSQL(mTable);

}
}
```

**Código 41. Creación de la base de datos SQLite.**

### Consulta, actualización e inserción de filas

Dentro de la clase `FriendDataBase` existen dos métodos que interactúan con la base de datos: `getFriends()` e `insertFriend()`. El primero devuelve todos los contactos y el segundo actualiza los datos de un contacto o, en caso de no existir, lo inserta. Seguidamente, se mostrará el código del segundo método al ser el más interesante de los dos:

```
public class FriendDataBase {

public void insertFriend(String number, String latitude, String
longitude, String date){

      // Consulta en la BD la existencia del número de teléfono
      Cursor c = mDataBase.query("friend", new String[] {
                        "number"},
                        "number="+number,
                        null,
                        null,
                        null,
                        null);

      // Crea una tupla completa para el contacto
      ContentValues values = new ContentValues();
      values.put("latitude", latitude);
      values.put("longitude", longitude);
      values.put("date", date);


      // No existe, insertar en la tabla
      if (c.getCount() == 0){
            values.put("number", number);
            mDataBase.insert("friend", null, values);
      }else{ // Sí existe, actualizar
            mDataBase.update("friend", values, "number="+number, null);
      }
}
}
```

**Código 42. Consulta, actualización e inserción de filas en la base de datos SQLite**

La primera acción es consultar la base de datos para conocer si existe el contacto al que se le van a actualizar los datos de localización. El método `query()` de la clase `SQLiteDatabase` permite lanzar una consulta, de forma que mediante sus parámetros se construye una sentencia SELECT completa: tabla a consultar, columnas que se desea devolver, posible cláusula WHERE y orden de las filas (ver el apartado 4.2.10). Esta consulta arrojará su resultado sobre un objeto de la clase `Cursor`.

En caso de devolver alguna fila, según indique el método `getCount()` de la clase `Cursor`, significa que el contacto ya existe y se puede actualizar con el método `update()` de `SQLiteDatabase`; si no devuelve ninguna, el contacto no existe y debe ser insertado con el método `insert()`.

En cualquier caso, es necesario agrupar de alguna forma los valores para la tupla, ya sea para su inserción o su actualización. Dentro del paquete `android.content` (aquel importante que contiene clases como `Intent` o `BroadcastReceiver`) existe una clase llamada `ContentValues` que permite asociar pares del tipo campo-valor perfectos para su utilización en bases de datos como SQLite. De esta forma, se compone un objeto `ContentValues` con la latitud, longitud y fecha del contacto para su utilización en `insert()` o `update()`.

**Cuándo guardar la información de los contactos en SQLite**

Como se recordará, la información de localización de los contactos se obtiene a través de conexiones periódicas con el servidor. Cada vez que se realiza una conexión exitosa, la lista con los contactos `mFriendList` se actualiza. En caso de no ser posible la conexión, se utiliza la información presente en la base de datos SQLite. Pero, ¿en qué momento se guarda esta información en la base de datos para ser recuperada en caso de necesidad?

La información de localización de los contactos se guardará desde la lista `mFriendList` a la base de datos SQLite siempre que se cierre la aplicación *ContactMap*, o cada vez que exista el riesgo de que la aplicación pueda ser eliminada por falta de recursos. De esta forma, se pretende asegurar que siempre se tendrá acceso a la última información de localización obtenida.

La clase principal `ContactMap`, como clase que deriva de Activity, tiene un ciclo de vida bien definido y gestionado por el propio Android, tal y como se explicó en un capítulo anterior. Cada transición entre estados de la actividad (en ejecución, en pausa, etc.) implica la llamada del método correspondiente.

Por ejemplo, al crearse por primera vez la actividad se lanza siempre el método `onCreate()`. Así mismo, cuando la actividad es eliminada de forma intencionada por el usuario, se llama al método `onDestroy()`. Una de las situaciones bajo la cual la actividad `ContactMap` puede ser eliminada por el sistema por falta de recursos es

cuando esta pasa a un segundo plano y, además, ya no es visible para el usuario. Tal circunstancia va precedida por la llamada al método `onStop()`.

Es en ambos estados del ciclo de vida, `onStop()` y `onDestroy()`, cuando se recorre la lista `mFriendList` con los contactos y se almacena, uno por uno, en la base de datos de SQLite para poder ser repuesta en caso de que no exista posibilidad de establecer conexión con el servidor. Cuando tal circunstancia se detecta, se utiliza el método `updateFriendsFromDataBase()` definido en `ContactMap`.

### 4.2.13 Construcción del menú principal

Para poder acceder a algunas de las opciones ofrecidas por *ContactMap*, el usuario debe utilizar el menú principal de la aplicación. Este menú aparece al pulsar la tecla correspondiente en el dispositivo móvil.

La clase `ContactMap` deriva de la clase `Activity` y tiene por ello un ciclo de vida bien definido. Además, cuenta con otra serie de métodos nativos que permiten tomar el control de diversos eventos provocados por el usuario. Este es el caso de la utilización del menú principal, cuyo comportamiento viene determinado por los métodos `onCreateOptionsMenu()` y `onMenuItemSelected()`.

#### Crear un menú

El método `onCreateOptionsMenu()` es invocado cuando se pulsa la tecla correspondiente al menú, y permite configurar el tipo de menú que se quiere desplegar. Este menú viene representado generalmente por una ventana que se desliza sobre la interfaz mostrada en ese momento y que debe ofrecer las funcionalidades principales de la aplicación en curso, que en el caso de *ContactMap* son las siguientes:

- Listar los contactos
- Cambiar el tipo de mapa
- Realizar una llamada al contacto
- Mandar un SMS al contacto
- Mandar un correo electrónico al contacto

En el siguiente fragmento de código se enseña al lector la forma en la que se compone el menú principal de *ContactMap*.

```
public class ContactMap extends MapActivity {

@Override
public boolean onCreateOptionsMenu(Menu menu) {

      // Crear las siguientes opciones de menú

      MenuItem item0 = menu.add(0, 0, 0, R.string.option_list);
      MenuItem item1 = menu.add(0, 1, 0, R.string.option_mapa);
      MenuItem item2 = menu.add(0, 2, 0, R.string.option_call);
      MenuItem item3 = menu.add(0, 3, 0, R.string.option_sms);
      MenuItem item4 = menu.add(0, 4, 0, R.string.option_email);

      // Asocia una imagen a cada opción de menú
      item0.setIcon(R.drawable.agenda);
      item1.setIcon(R.drawable.mapmode);
      item2.setIcon(R.drawable.call);
      item3.setIcon(R.drawable.sms);
      item4.setIcon(R.drawable.email);

      return super.onCreateOptionsMenu(menu);
}
}
```

**Código 43. Construcción del menú principal**

La clase `Menu` representa el menú principal de una aplicación y forma parte del paquete `android.view`, que ofrece un considerable número de clases para manejar los elementos de una interfaz de usuario y gestionar los eventos asociados. A través del objeto `Menu` se gestionan las opciones que componen el menú principal.

Un clase `MenuItem` (también del paquete `android.view`) representa una opción dentro del menú y permite tener acceso a sus propiedades. Cada uno de estos objetos se crea a partir del método `add()` de `Menu`, donde se especifica entre otras cosas el orden que debe ocupar la aplicación dentro del menú y el texto que la debe acompañar.

Las opciones declaradas para el menú principal cuentan con un texto asociado que las describe y una imagen o icono representativo, ambos visibles para el usuario cuando despliegue el menú. Para lo primero, el texto, se incluye en la propia llamada a `Menu.add()` y se utiliza el texto declarado en el fichero de recursos "strings.xml", dentro de la carpeta "\res\values" de la aplicación.

Para las imágenes, se pueden asociar a cada opción a través del método `setIcon()` de la clase `MenuItem`. Al igual que con el texto, las imágenes se incluyen haciendo una referencia a la carpeta de recursos "\res\drawable", donde deberán estar ubicadas. Como información de utilidad al lector, se dirá que el SDK de Android incluye por defecto una importante colección de iconos útiles para todo tipo de aplicaciones. Estos iconos se encuentran disponibles en la carpeta "\tools\lib\res\default\drawable".

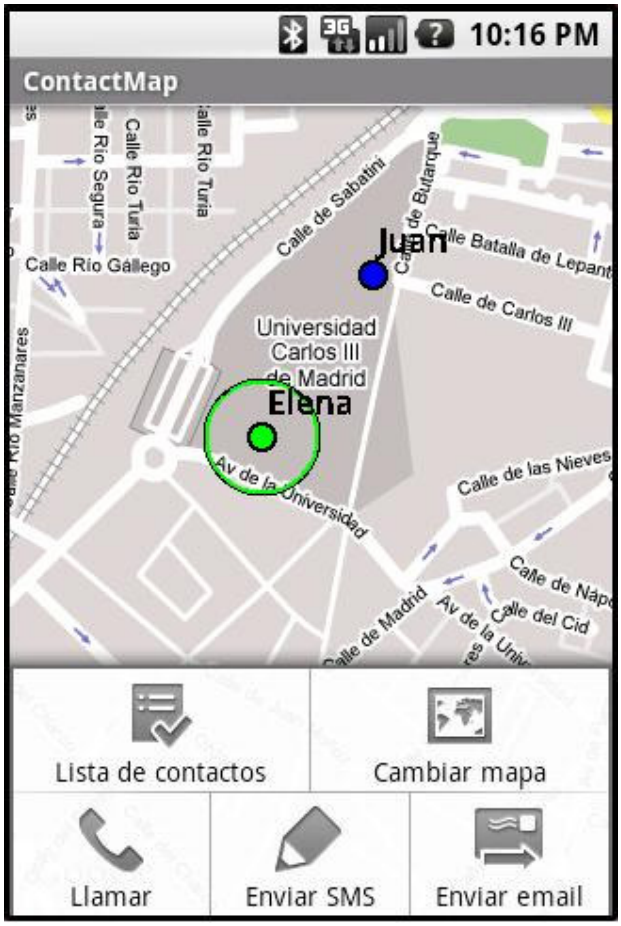


**Figura 27. Despliegue del menú principal**

### Controlar la opción seleccionada

En este punto, el usuario ya puede desplegar el menú y comprobar que opciones principales le ofrece `ContactMap`.Ahora es necesario poder conocer qué opción pulsa el usuario y actuar en consecuencia.

El otro método importante para controlar el menú y sus opciones es `onMenuItemSelected()`. Cada vez que el usuario selecciona una opción del menú principal, este método se llama recibiendo como parámetro un objeto `MenuItem` que va a permitir conocer qué opción exactamente ha sido la pulsada.

Utilizando una sencilla sentencia de control switch y el método `getItemId()` de `MenuItem`, se obtiene la opción pulsada y se ejecuta la acción correspondiente a la misma, tal y como se hace en el Código 44.

```
public class ContactMap extends MapActivity {

@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {

      switch (item.getItemId()) {

            case 0:// Listar contactos
                  // (...)
                  return true;

            case 0:// Cambiar mapa
                  // (...)
                  return true;

            case 0:// Llamar
                  // (...)
                  return true;

            case 0:// Enviar SMS
                  // (...)
                  return true;

            case 0:// Enviar correo electrónico
                  // (...)
                  return true;
      }

      return super.onMenuItemSelected(featureId, item);
}

}
```

**Código 44. Control del menú principal**

### 4.2.14 Mostrar listado de contactos

Acciones como mandar un SMS, un correo electrónico o realizar una llamada se realizan sobre el contacto que en ese momento tenga el foco de la aplicación. El usuario puede cambiar el foco de un contacto a otro de diferentes maneras: pulsando sobre ellos, utilizando los botones de anterior y siguiente (se verán más adelante) o a través de un listado con todos los contactos ordenados alfabéticamente. Éste último caso es el que se explica en este apartado.

Cualquier aplicación en Android está formada por una serie de componentes básicos entre los que se cuenta *Activity*. Una clase `Activity`, como ya sabrá el lector, representa una funcionalidad importante, con entidad propia, y que está asociada generalmente a una interfaz con la que el usuario puede interactuar. La clase `ContactMap`, por ejemplo, es una *Activity* que muestra un mapa y permite al usuario realizar una serie de acciones. Una *Activity* también puede invocar a su vez a otros componentes *Activity*.

El listado de contactos consiste en una lista a pantalla completa donde se ordenan alfabéticamente todos los contactos y donde el usuario pulsa sobre uno de ellos; se vuelve entonces de forma inmediata al mapa, donde dicho contacto adquiere el foco de la aplicación. Toda esta acción se enmarca dentro de un nuevo componente *Activity*, ya que representa una funcionalidad importante con su propia y diferenciada interfaz.

### Lanzar una nueva Activity

La clase `FriendListViewer` extiende a la clase `Activity` y es la encargada de componer la lista de contactos y capturar cuál de ellos es pulsado. En primer lugar, a través del Código 45 se enseña cómo se lanza esta nueva *Activity* desde el método `onMenuItemSelected()` que, como se recordará, es el método invocado al seleccionar una opción del menú principal.

```
public class ContactMap extends MapActivity {

@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {

      switch (item.getItemId()) {

            case 0:// Listar contactos

                  // Intent asociado a la clase FriendListViwer
                  Intent listar = new Intent
                        (this, FriendListViewer.class);

                  i = 1;

                  // Recorrer contactos
                  // Cada nombre será añadido al Intent como extra
                  while (mFriendList.hasNext()){
                        listar.putExtra("contact"+i.toString(),
                                          mFriendList.getName());
                        i++;
                  }

                  // Lanzar Activity (FriendListViewer)
                  this.startActivityForResult(listar,0);

                  return true;

            // (...)
      }
      return super.onMenuItemSelected(featureId, item);
}
}
```

**Código 45. Lanzar *Activity* para el listado de contactos**

Toda nueva *Activity* debe ser lanzada a través de un *Intent*, que especifica qué es lo que se quiere hacer y con qué datos debe hacer. Habitualmente, un *Intent* es una forma de delegar determinadas acciones en otras aplicaciones instaladas en el sistema y que, obviamente, tengan la capacidad para llevarlas a cabo. Sin embargo, también es posible especificar que un *Intent* debe ser realizado por una determinada instancia de una clase. Es lo que se pretende hacer en este caso con el listado de contactos.

Al crear un objeto de la clase `Intent` se especifica en el constructor que dicho *Intent* debe ser realizado en la clase `FriendListViewer`. De esta forma, ya se ha configurado qué es lo que se quiere hacer (ejecutar la clase `FriendListViewer`). Ahora se debe indicar con qué datos se desea hacer. Ya que el objetivo final es mostrar un listado con los contactos, lo que se va a pasar como datos asociados a dicho *Intent* es el nombre de cada uno de los contactos presentes en la lista `mFriendList`.

La clase `Intent` permite asociar muchos tipos básicos de datos para poder ser procesados en la clase o aplicación donde vaya a ser atendido el *Intent*. Por ejemplo, en esta situación concreta se ha utilizado el método `putExtra()` para asociar cadenas de texto correspondientes a cada uno de los nombres. Una peculiaridad común a cada dato asociado a un *Intent*, cualquiera que sea su tipo, es que debe ir acompañado de una etiqueta textual que lo identifique. Por ello, al método `putExtra()` se le asigna tanto el nombre del contacto como la etiqueta identificativa "contactoX", donde la X se sustituye por el orden que debe ocupar en la lista.

El *Intent* ya ha sido construido: representa una acción específica (ejecutar la clase `FriendListViewer`) y unos datos asociados (los nombres de los contactos a listar). Únicamente resta lanzar la *Activity*.

Existen dos formas para lanzar una nueva actividad en Android. Si se quiere lanzar la *Activity* sin más, sin esperar ningún resultado final, se utiliza el método `startActivity()`. Si se espera un valor devuelto por la *Activity*, entonces se lanza con el método `startActivityForResult()`, de forma que al terminar dicha actividad se invocará el método `onActivityResult()`, que permite manejar el resultado obtenido. Como en este caso se desea conocer qué contacto ha sido seleccionado en la lista, se utilizará el segundo de los métodos posibles.

Invocar el método `startActivityForResult()` es muy sencillo. Simplemente se indica el *Intent* que sea desea lanzar y se le asocia un código identificativo para poder diferenciar en `startActivityForResult()` cuál es la *Activity* que ha finalizado. Es decir, todas las actividades lanzadas con `startActivityForResult()` acaban en el mismo método `onActivityResult()`, que ha de poder diferenciarlas de algún modo.

**Mostrar el listado de contactos**

Al lanzar el *Intent* explicado anteriormente, la clase `ContactMap` deja de ser la actividad en curso para entrar en un estado `onPause()` primero y `onStop()` después,

ya que no es visible para el usuario. La clase `FriendListViewer`, que también extiende a `Activity`, toma toda la pantalla ofreciendo al usuario un listado de contactos.

Esta clase debe extraer los datos asociados al *Intent* con el que ha sido lanzada, componer la lista de nombres y capturar cuál de ellos ha sido pulsado, para poder retornar ese resultado a la clase `ContactMap` (que despertará y pasará a un estado `onResume()`) y su método `onActivityResult()`.

En primer lugar, se muestra el método `onCreate()` de `FrienListViewer`, que extraerá los datos asociados al *Intent* y compondrá la lista de contactos. El Código 46 enseña el proceso.

```
public class FriendListViewer extends ListActivity {


// Lista para los nombres
public ArrayList<String> mFriendShowList = new ArrayList<String>();

@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);



      // Obtener los extras incluídos en el Intent
      Bundle extras = getIntent().getExtras();

      // Obtener los nombres
      Integer i=0;
      while (i<extras.size()){
            mFriendShowList.add(
                  extras.getString("contact"+i.toString()));
            i++;
      }

      // Adapta la lista al formato preciso para ser mostrada
      setListAdapter(new ArrayAdapter<String>(this,
      android.R.layout.simple_list_item_1, mFriendShowList));

      this.setSelection(1);

}
```

**Código 46. Mostrar listado de contactos**

Lo primero de todo es mencionar que, tal y como muestra el código anterior, la clase `FriendListViewer` no extiende directamente a la clase `Activity`, sino que extiende a `ListActivity`. Esta clase, al igual que ocurría en la clase `ContactMap` con

`MapActivity`, es derivada de `Activity` y, si la primera estaba específicamente adaptada para gestionar mapas, la segunda lo está para mostrar listados a pantalla completa donde el usuario ha de elegir uno de los elementos. Además, la clase `ListActivity` tiene asociado un objeto `ListView`, que representa uno de los muchos diseños para interfaces de usuario predefinidas en Android y que ayuda en la tarea de componer y mostrar un listado vertical de elementos.

La clase `Bundle` forma parte del paquete `android.os`, un pequeño pero relevante paquete que ofrece varios servicios asociados al sistema operativo, paso de mensajes o comunicaciones entre procesos. Esta clase representa una colección de valores y suele utilizarse para recuperar los datos asociados a un *Intent*, tal y como se hace en `FriendListViewer`.

Mediante el método `getIntet()` de la clase `Activity` se obtiene el objeto `Intent` con el que fue lanzada dicha actividad (si es el caso). A su vez, este objeto `Intent` ofrece un método llamado `getExtras()` que devuelve el conjunto total de valores asociados a él. A través del objeto `Bundle`, que contendrá todos los valores del *Intent*, se van recorriendo uno a uno y se recuperan individualmente accediendo a ellos a través de su etiqueta identificativa. Al mismo tiempo que se recuperan, se van almacenando en una nueva lista de tipo *string* llamada `mFriendShowList`. En esta lista es donde estarán guardados finalmente todos los nombres de contactos que se van a mostrar en el listado al usuario.

`ListActivity` tiene un método exclusivo al resto de clases derivadas de *Activity* llamado `setListAdapter()` que construye con los valores dados el listado final que será visible para el usuario a pantalla completa. Para ello, es necesario transformar antes la lista `mFriendShowList` en un objeto de la clase `ListAdapter`.

Por último, con `setSelection()` de `ListActivity` se especifica cuál de los elementos de la lista debe aparecer como seleccionado por defecto al usuario; en este caso, es el primero de ellos.

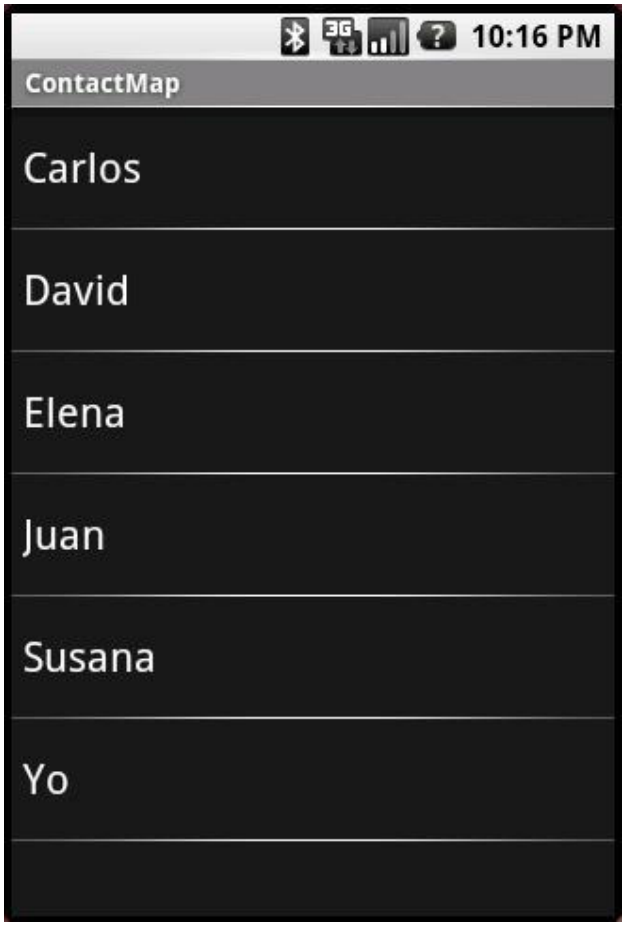


**Figura 28. Listado de contactos**

Ya se ha construido y mostrado el listado al usuario, pero ahora es necesario saber qué hacer cuando el usuario seleccione uno de los nombres. El método `onListItemClick()` de `ListActivity` se activa cuando tal evento sucede, y debe devolver a `ContactMap` la posición del usuario seleccionado con `setResult()`, además de dar por terminada con el método `finish()` la presente actividad `FriendListViewer`.

```
public class FriendListViewer extends ListActivity {

@Override
protected void onListItemClick(ListView l, View v, int position, long
id) {

      super.onListItemClick(l, v, position, id);

      // Devuelve la posición del elemento seleccionado
      this.setResult(position);


      // Terminar la presente Activity
      this.finish();

}
}
```

**Código 47. Devolver elemento seleccionado en la lista de contactos**

### Manejar el resultado de una Activity

Ya que la *Activity* encarnada en la clase `FriendListViewer` se lanzó con el método `startActivityForResult()`, cuando esta termina es invocado de inmediato el método `onActivityResult()` declarado en la clase `ContactMap`.

De esta forma, es posible gestionar el resultado proporcionado por una *Activity*. Una vez finalizada, dicho método es llamado proporcionando acceso tanto al resultado final como al código identificativo con el que fue lanzado la actividad, ya que es el que va a permitir a este método conocer cuál es la actividad finalizada.

Una vez el usuario ha seleccionado un contacto de la lista mostrada, la aplicación debe establecer este contacto como el poseedor del foco de aplicación, centrando además el mapa en él y haciéndole objeto de otras acciones posterior, como el envío de un SMS, una llamada telefónica, o el envío de un correo electrónico.

El Código 48 muestra este proceso:

```
public class ContactMap extends MapActivity {

// Contacto con el foco de la aplicación
public Friend mCurrent = null;
// Lista de contactos
public ArrayList<Friend> mFriendList = new ArrayList<Friend>();
```

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {

      super.onActivityResult(requestCode, resultCode, data);

      // Código correspondiente a la actividad FriendListViewer
      if (requestCode==0){

            // Centrar mapa en el contacto seleccionado
            mMapControl.animateTo(
                  mFriendList.get(resultCode).getGeo());

            // Cambiar contacto con el foco
            mCurrent=mFriendList.get(resultCode);

      }
      // (...)
}

}
```

**Código 48. Contacto seleccionado por el usuario**

Un último apunte que se debe mencionar con respecto a la clase `FriendListViewer` es que, como `Activity` que es, debe ser declarada explícitamente en el fichero "AndroidManifest.xml". Recuérdese que cualquiera de los componentes básicos de Android que se utilicen en una aplicación debe ser declarado para poder utilizarlo. Así pues, la actividad queda declarada de la siguiente manera dentro del elemento `<application>`:

```
<activity android:name=".FriendListViewer"></activity>
```

**Código 49. Declaración en el manifiesto de la *Activity* FriendListViewer**

### 4.2.15 Llamar a un contacto

Entre las acciones que el usuario puede realizar sobre un contacto está la de realizar una llamada telefónica. Para tal fin se utilizará un *Intent* y se lanzará una nueva *Activity*, pero de una forma algo diferente a lo recientemente visto con el listado de los contactos.

Como ya se ha explicado en anteriores ocasiones, mediante un *Intent* se delega una determinada acción en el sistema, de forma que este encuentra la aplicación más

apropiada de entre todas las instaladas para llevarla a cabo. Un *Intent* consiste en la especificación de la acción a realizar, y los datos con los que debe realizarse.
Android cuenta por defecto con un gestor de llamadas telefónicas. Esto permite poder lanzar el *Intent* específico de una llamada sin tener que preocuparse de nada más, puesto que el sistema la derivará a dicho gestor de llamadas de Android.

El siguiente código, Código 50, expone los pasos a seguir para lanzar un *Intent* de llamada telefónica. La acción se lleva a cabo en el método `onMenuItemSelected()` de la clase `ContactMap`, una vez que el usuario ha seleccionado la opción correspondiente en el menú principal.

```
public class ContactMap extends MapActivity {

@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {

      switch (item.getItemId()) {

            case 2: // Llamar al contacto con el foco

                  // Crear Intent para llamada telefónica
                  Intent llamar = new Intent(Intent.ACTION_CALL);

                  // Asociar al Intent el número de teléfono
                  Uri uri=Uri.parse("tel:"+mCurrent.getNumber());
                  llamar.setData(uri);

                  // Lanzar Activity que atienda este Intent
                  this.startActivity(llamar);

                  return true;

                  // (...)
      }
}
```

**Código 50. *Intent* para realizar una llamada telefónica.**

Al crear el objeto `Intent` se especifica el tipo de acción que se desea realizar. En el caso del listado de contactos se indicaba que se quería ejecutar una clase concreta que llevaba a cabo dicha acción (la clase `FriendListViewer`). Aquí también se indica la acción a realizar, pero no diciendo directamente quién debe realizarla, sino diciendo qué se quiere realizar. El sistema ya se encargará de buscar la aplicación más indicada para ello.

Existen acciones frecuentes y definidas para indicar en un *Intent*. La correspondiente a la llamada telefónica es `Intent.ACTION_CALL`. Una vez creado el objeto `Intent` es necesario añadir como datos el número de teléfono al que se desea llamar; este debe ser

adjuntado como un objeto `Uri`. Las URI relacionadas con recursos telefónicos han de tener el formato "tel:numero" para poder funcionar correctamente con la acción `Intent.ACTION_CALL`.

Tras la creación y configuración del objeto `Intent` ya se puede lanzar la actividad. Como en este caso no se espera que esta retorne ningún valor que deba ser capturado, la *Activity* se lanza sin más con el método `startActivity()`. En ese momento el sistema abrirá el gestor de llamadas y marcará el número indicado.

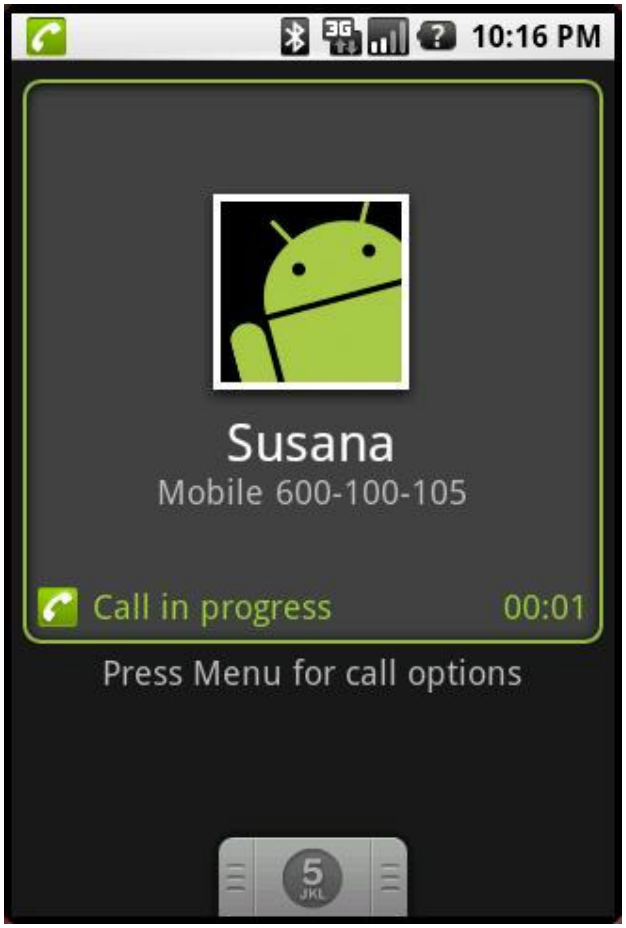


**Figura 29. Llamada a un contacto**

El acceso a las funciones telefónicas del dispositivo móvil requiere de su correspondiente permiso en la declaración del manifiesto. El permiso es el siguiente:

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

**Código 51. Declaración en el manifiesto del permiso de llamada telefónica**

### 4.2.16 Enviar un correo electrónico

Otras de las funcionalidades ofrecidas por *ContactMap* es la de enviar un correo electrónico a un contacto. Al igual que ocurre con la llamada telefónica, esta acción se llevará a cabo a través de un *Intent* de forma que sea el sistema el que se haga cargo a través de la aplicación más adecuada.

```
public class ContactMap extends MapActivity {

@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {

      switch (item.getItemId()) {

            case 4: // Enviar email al contacto con el foco

                  // Crear Intent para enviar email
                  Intent enviarEmail = new Intent(Intent.ACTION_SEND);

                  // Asociar al Intent la dirección y el tipo de dato
                  enviarEmail.putExtra(
                              Intent.EXTRA_EMAIL,mCurrent.getEmail());
                  enviarEmail.setType("message/rfc822");

                  // Lanzar Activity que atienda este Intent
                  this.startActivity(Intent.createChooser(
                                          enviarEmail,
                                          "Enviar un email a"
                                          +mCurrent.getName()));

                  return true;
                  // (...)
      }
}
```

**Código 52. *Intent* para enviar un correo electrónico**

El Código 52 enseña la parte del método `onMenuItemSelected()` de la clase `ContactMap` que permite enviar un correo electrónico. En primer lugar se crea el

objeto `Intent` correspondiente, en este caso asociándole la acción `Intent.ACTION_SEND`.

Después, como datos adjuntos a este *Intent* se incluyen tanto la dirección de correo electrónico del contacto actualmente con el foco, como el tipo MIME del correo. A continuación se lanza la *Activity* correspondiente con el método `startActivity()`, ya que no se precisa manejar ningún resultado devuelto por ésta.

Una peculiaridad de esta llamada a `starActivity()` es que se utiliza como parámetro el método `createChooser()` de la clase `Intent`. Este permite que, en caso de existir varias aplicaciones capaces de hacerse cargo de dicho *Intent*, el sistema muestre al usuario un menú donde puede elegir la aplicación para desarrollar la acción.

Si en el caso de las llamadas telefónicas existe por defecto un gestor de las mismas en Android, para el correo electrónico no hay instalado ningún cliente. Es por ello que, en la versión considerada de Android para este proyecto, este *Intent* no podrá ser atendido por ninguna aplicación, hecho que el sistema comunicará al usuario mediante el correspondiente mensaje.

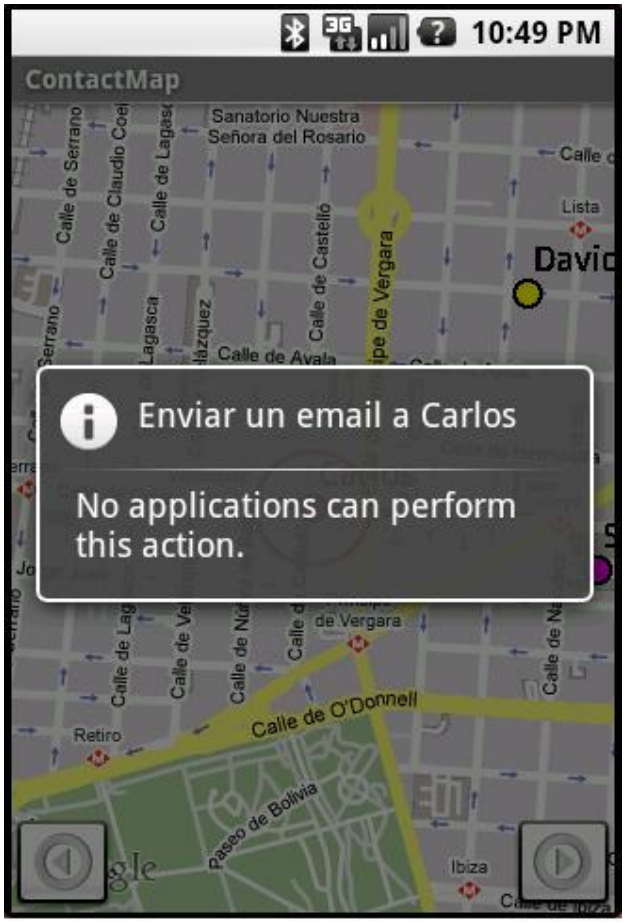


**Figura 30. Mensaje tras intentar enviar un correo electrónico**

### 4.2.17 Enviar un SMS

El proceso de envío de un SMS es bastante similar al visto en el listado de contactos, en el apartado 4.2.14. No se utilizará un *Intent* para delegar en el sistema dicha operación, sino que nuevamente se utiliza para lanzar una nueva *Activity* que se corresponde con una clase ya implementada en la propia aplicación *ContactMap*, de nombre `SMSWriter`.

Esta clase muestra al usuario una nueva interfaz donde poder escribir el mensaje de texto, tal y como exhibe la Figura 31. La interfaz consta de los siguientes elementos:

- Una caja de texto para introducir el número, acompañada de su correspondiente etiqueta textual.
- Una caja de texto para introducir el mensaje, acompañada igualmente de su etiqueta textual.
- Un botón para enviar.
- Un botón para borrar.

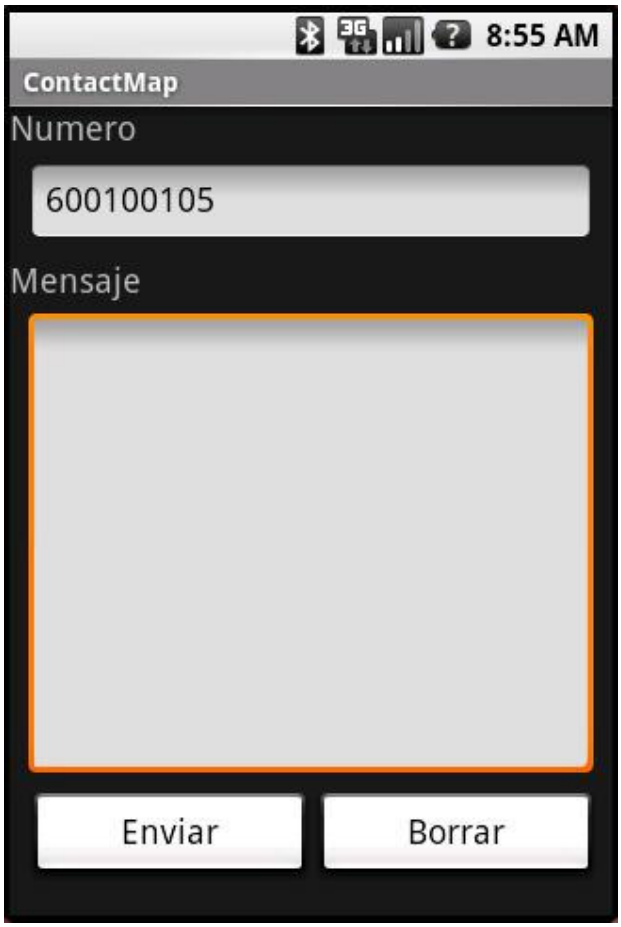


**Figura 31. Interfaz para enviar un SMS**

### Lanzar una nueva Activity

El lanzamiento de una nueva *Activity* sigue siendo igual al visto en anteriores ocasiones. En el método `onMenuItemSelected()` de la clase `ContactMap`, que es llamado cada vez que el usuario selecciona una opción del menú principal, se crea un nuevo *Intent* y se lanza la *Activity* correspondiente a este.

El objeto de la clase *Intent* está asociado a la clase `SMSWriter`, debido a que no se quiere que el sistema se haga cargo y busque la aplicación instalada más adecuada, sino que se desea explícitamente que se ejecute en esta clase implementada en la propia aplicación *ContactMap*.

Como datos asociados al *Intent* (recuérdese, un *Intent* consta de una acción y unos datos) se encuentra únicamente el número de teléfono del contacto que cuenta en ese momento con el foco, de forma que ya aparezca ese campo rellenado en la interfaz del usuario.

Es necesario que la nueva *Activity*, una vez finalizada, devuelva a la *Activity* principal, `ContactMap` el texto introducido por el usuario para poder enviar el mensaje. Por ello se lanza con el método `startActivityForResult()`, que llamará a `onActivityResult()` cuando la actividad termine.

En el siguiente código se muestra cómo lanzar la nueva *Activity* desde el método `onMenuItemSelected()`:

```
public class ContactMap extends MapActivity {
@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {

      switch (item.getItemId()) {
            case 3: // Enviar SMS al contacto actual

                  // Crear Intent asociado a la clase SMSWriter
                  Intent mandarSMS = new Intent(this, SMSWriter.class);

                  // Asociar al Intent el número del amigo actual
                  mandarSMS.putExtra("number", mCurrent.getNumber());

                  // Lanzar Activity (SMSWriter) que atienda al Intent
                  this.startActivityForResult(mandarSMS,1);

                  return true;
            }
            // (...)
      }
}
```

**Código 53. Lanzar *Activity* para enviar un SMS**

**Crear la interfaz a partir de XML**

Android ofrece la posibilidad, ya vista en anteriores apartados, de utilizar en el código fuente los recursos externos declarados dentro de la carpeta "\res". Hasta ahora se han visto casos donde se declaraban externamente cadenas de texto para las opciones del menú principal o ciertos avisos al usuario, facilitando así su utilización en otros idiomas; también se ha enseñado como utilizar imágenes o iconos que después pueden ser referenciadas en el código con una simple declaración que apunte a esta carpeta de recursos externos.

Sin embargo, quizás la mayor utilidad de estas declaraciones externas al código resida en la posibilidad de definir, de nuevo a través de documentos XML, interfaces de usuario completas. Dentro de la carpeta "\res\layout" pueden definirse cuantos documentos XML se desee, cada uno de ellos representando una determinada interfaz que se quiera implementar, o incluso diferentes versiones de la misma para según qué casos.

Para la interfaz de usuario que debe mostrarse con la clase `SMSWriter` se utilizan estas declaraciones externas, ubicadas en el fichero de nombre "sms.xml". Dada la Figura 31 anteriormente enseñada, sabemos que hemos de utilizar los siguientes elementos del paquete `android.widget` que, como se vio anteriormente, contiene numerosas clases para construir interfaces de usuario:

- Para las etiquetas textuales, la clase `TextView`.
- Para las cajas de texto, la clase `EditText`.
- Para los botones, la clase `Button`.

Cada una de estas clases, ya sean instanciadas desde un documento XML externo o en el código fuente de forma tradicional, tienen una serie de atributos que han de ser configurados y que matizan su aspecto, colocación o comportamiento dentro de la interfaz.

El siguiente ejemplo escrito en XML define dos de los elementos presentes en la interfaz para escribir un SMS. Son la etiqueta y la caja de texto correspondiente a la parte donde el usuario puede introducir el contenido del mensaje que desea escribir.

```
<TextView
android:id="@+id/txt2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginTop="5px"
android:text="Mensaje"
android:textSize="18sp"
/>
```

```
<EditText
android:id="@+id/message"
android:layout_width="300sp"
android:layout_height="250sp"
android:layout_marginTop="5px"
android:layout_marginLeft="10px"
android:textSize="18sp"
android:maxLength="160"
/>
```

**Código 54. Construcción con XML de elementos de interfaz de usuario**

Cada elemento declarado tiene como nombre el propio de la clase a la que pretende instanciar, en este caso `<TextView>` y `<EditText>`. Dependiendo de dicha clase, los elementos podrán declarar unos atributos u otros, aunque aquí los atributos son muy similares entre ambos. Se pueden observar los siguientes atributos:

- `android:id`: todo elemento declarado debe tener un identificador único, escrito en el atributo `android:id`. Este identificador es a través del cual se referencia el elemento en el código fuente, y es el nombre con el que aparece declarado el recurso dentro del fichero "R.java" (se recordará que dicho fichero mantiene una sincronización con cualquier elemento declarado como recurso en la carpeta "\res", de forma que permite a las demás clases de la aplicación Android poder utilizar el recursos deseado simplemente nombrándolo).

- `android:layout_width`: indica la anchura que debe tener este elemento. Si se especifica `wrap_content`, entonces ocupará tanto espacio como necesite; por otra parte, una configuración como `fill_parent` indica al *widget* que ocupe tanto espacio como tenga disponible. Otra alternativa, como ocurre en `<EditText>` es especificar exactamente el tamaño deseado, ya sea en píxeles (`px`), en píxeles escalados (`ps`), en pulgadas (`in`), milímetros (`mm`) u otros.

- `android:layout_height`: indica la altura del elemento. Se comporta igual que el atributo `layout_width`.

- `android:layout_marginTop`: especifica el margen superior que se desea establecer para el elemento. Se comporta igual que el atributo `layout_width`.

- `android:layout_marginLeft`: especifica el margen izquierdo que se desea establecer para el elemento. Se comporta igual que el atributo `layout_width`.

- `android:Text`: el texto que, por defecto, debe mostrar el elemento.

- `android:textSize`: el tamaño del texto.

- `android:maxLength`: el número máximo de caracteres.

Cada elemento de interfaz cuenta con su propia familia de atributos, dando así un alto grado de libertad al desarrollador para configurar sus *widgets* de la forma que considere más oportuna.

Además de elementos de interfaz, Android también contempla algunas vistas, también llamados diseños, que agrupan los distintos *widgets* y los ordenan siguiendo unos patrones frecuentemente utilizados en las aplicaciones. En ocasiones anteriores hemos nombrado `Gallery`, para los visores de fotografías, o `ListView`, utilizado aquí para mostrar un listado de contactos. Para ordenar adecuadamente los elementos de la interfaz de `SMSWriter` se usará el patrón llamado `LinearLayout`.

La clase `LinearLayout` permite colocar los elementos de una interfaz de forma consecutiva, ya sea imaginando para ellos unas columnas en la pantalla (orden horizontal) o unas filas (orden vertical). Este patrón se utiliza en dos ocasiones dentro del fichero "sms.xml", donde estamos definiendo los *widgets*: en una ocasión para la totalidad de los elementos, siguiendo un orden vertical, y en otra ocasión para colocar los dos botones de envío y borrado siguiendo un orden horizontal. La declaración de este patrón es la mostrada en el Código 55, donde se indican sólo a través de su nombre los elementos completos de la interfaz:

```
<LinearLayout
android:id="@+id/linearlayout1"
android:orientation="vertical"
>

<TextView android:id="@+id/txt1" … />

<EditText android:id="@+id/number" … />

<TextView android:id="@+id/txt2" … />

<EditText android:id="@+id/message" … />

<LinearLayout
android:id="@+id/linearlayout1"
android:orientation="horizontal"
>

<Button android:id="@+id/button1" … />

<Button android:id="@+id/button2" … />

</LinearLayout>

</LinearLayout>
```

**Código 55. Uso del diseño LinearLayout**

### Mostrar la interfaz desde la clase `SMSWriter`

Una vez declarada completamente la interfaz de usuario en un documento XML dentro de la carpeta "\res\layout", solamente es necesario referenciarla dentro del código fuente de `SMSWriter`. El método `setContentView()` de la clase `Activity` es la única llamada que es necesaria hacer para construir y mostrar completamente al usuario la interfaz. El elemento que la referencia, en el caso aquí descrito, es `R.layout.sms`.

Sin embargo, si se desea hacer un control adicional de algunos de los elementos como, por ejemplo, captar sus eventos asociados, es necesario instanciarlos individualmente para tener un objeto que poder controlar. Esta instanciación es tan simple como la anterior. El siguiente código muestra cómo se referencia la interfaz completa y se instancia alguno de sus elementos para poder controlarlos:

```
public class SMSWriter extends Activity{

@Override
protected void onCreate(Bundle savedInstanceState) {

      super.onCreate(savedInstanceState);

      // Construir interfaz completa
      setContentView(R.layout.sms);

      // Instanciar algunos elementos de la interfaz
      EditText editNumber = (EditText) findViewById(R.id.number);
      EditText editSMS = (EditText) findViewById(R.id.message);
      Button button1 = (Button) findViewById(R.id.button1);
      Button button2 = (Button) findViewById(R.id.button2);

      // Obtener número del destinatario
      Bundle extras = getIntent().getExtras();

      // Mostrarlo dentro de la caja de texto
      editNumber.setText(extras.getString("number"));
      editNumber.setFocusable(false);

      // ...

}
}
```

**Código 56. Utilizar recursos XML para instanciar la interfaz de usuario**

El método `findViewById()`, de la clase `Activity`, permite instanciar de forma inmediata cualquier elemento de interfaz declarado en la carpeta de recursos. Cada instancia se realiza haciendo un casting a la clase correspondiente y utilizando el

identificador declarado en el XML. De esta forma, ya se puede trabajar con los objetos obtenidos.

Mediante `getIntent().getExtras()` se obtienen los datos asociados al *Intent* que ha lanzado la presente *Activity*; el único dato asociado aquí es el número de teléfono del contacto al que se va a enviar el SMS. Este número se muestra en la caja de texto correspondiente con `setText()` de la clase `EditText`, y además se configura para que no pueda ser modificado, mediante el método `setFocusable()` y el valor `false`.

El siguiente control de elementos que se desea hacer es establecer los comportamientos para los dos botones mostrados, el de enviar y el de borrar. Al pulsar el botón de enviar, la *Activity* debe finalizar y retornar a la clase `ContactMap` el texto introducido por el usuario. Por su parte, al pulsar el botón de borrar, la caja de texto debe vaciarse por completo.

Para configurar estos comportamientos se implementa el método `onClick()` de cada botón. Esta implementación se hace, tal y como muestra el Código 57, a través de un listener llamado `setOnClickListener()`.

```
public class SMSWriter extends Activity{

@Override
protected void onCreate(Bundle savedInstanceState) {

      // (...)

      // Botón de enviar
      button1.setOnClickListener(new View.OnClickListener() {

            public void onClick(View view) {
                  // Añadir el texto del SMS a los datos del Intent
                  getIntent().putExtra(
                        "message", editSMS.getText().toString());
                  setResult(RESULT_OK,getIntent());
                  // Finalizar Activity
                  finish();
            }
      });

      // Botón de borrar
      button2.setOnClickListener(new View.OnClickListener() {
            public void onClick(View view) {
                  // Borrar texto
                  editSMS.setText("");
            }
      });
}
```

**Código 57. Comportamiento de los botones en la clase SMSWriter**

Para devolver a `ContactMap` el texto que ha introducido el usuario, se añade este como datos asociados al *Intent*. Además, se invoca el método `finish()` para dar por terminada la presente actividad. En caso de pulsar el botón de borrar, el texto introducido hasta el momento se borra utilizando `setText()` y la cadena vacía.

### Enviar el contenido del mensaje

Al termina la actividad `SMSWriter`, el control vuelve a `ContactMap` desde el método `onActivityResult()`. Desde esta clase, una vez recibido el texto que el usuario ha introducido, se envía el SMS de la forma que sigue:

```
public class ContactMap extends MapActivity {

// Contacto con el foco de la aplicación
public Friend mCurrent = null;
// Lista de contactos
public ArrayList<Friend> mFriendList = new ArrayList<Friend>();

@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {

      super.onActivityResult(requestCode, resultCode, data);

      // Resultado de la Activity SMSWriter
      if (requestCode==1){

            try{

            // Obtener gestor de SMS
            SmsManager smsManager = SmsManager.getDefault();

            // Enviar SMS con los datos devueltos
            smsManager.sendTextMessage(
            data.getExtras().getString("number"),
            null,
            data.getExtras().getString("message"),
            null,
            null);

            // Mensaje de confirmación
            Dialog enviado = new Dialog(this);
            enviado.setTitle(R.string.successSMS);
            enviado.show();
            enviado.setCanceledOnTouchOutside(true);

            }catch(Exception e){
```

```
            // Mensaje de error
            Dialog enviado = new Dialog(this);
            enviado.setTitle(R.string.failSMS);
            enviado.show();
            enviado.setCanceledOnTouchOutside(true);
            }

      }
}
}
```

**Código 58. Enviar SMS**

La clase `SmsManager`, presente en el paquete `android.telephony.gsm`, representa un gestor para los mensajes de texto. En particular, dispone del método `sendTextMessage()` para poder enviar mensaje de texto de tipo estándar. En este método, se especifican el número destino, el cuerpo del mensaje y el proveedor de servicios SMS, entre otros. Para indicar el número y mensaje, se obtienen del *Intent* los datos asociados. El proveedor de servicios SMS se especifica como `null`, lo cual significa que se utilizará el proveedor por defecto asociado a la tarjeta SIM presente en el dispositivo.

Para comunicar al usuario el resultado de tal operación, se utiliza la clase `Dialog` del paquete `android.app`. Mediante un objeto de esta clase, se advierte al usuario del éxito del envío o de algún error que haya tenido lugar durante el mismo. Como se puede comprobar, una vez más se utilizan los recursos textuales declarados en la carpeta "\res\values", dentro del fichero "strings.xml".

No hay que olvidar que para que la clase `SMSWriter` y el envío del SMS funcionen correctamente, es necesario hacer antes algunas declaraciones en el fichero del manifiesto. Por un lado, debe adjuntarse un permiso específico para los mensajes de texto:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

**Código 59. Declaración en el manifiesto del permiso para enviar SMS**

Además, ya que la clase `SMSWriter` es una *Activity*, esta debe ser declarada en el mismo "AndroidManifest.xml" como componente básico de la aplicación que es. La declaración mostrada en el debe hacerse dentro del elemento `<application>`.

```
<activity android:name=".SMSWriter"></activity>
```

**Código 60. Declaración en el manifiesto de la *Activity* SMSWriter**

### 4.2.18 Actualización periódica en el servidor

Anteriormente se ha explicado cómo la clase `HTTPConnection` se encarga de los aspectos de más bajo nivel del protocolo HTTP relacionados con las conexiones con el servidor; también se ha mencionado que las clases `XMLExchange` y `MyXMLHandler` se utilizan para procesar las peticiones enviadas y las respuestas recibidas en formato XML. Sin embargo, hasta ahora no se ha mencionado nada con respecto a cómo *ContactMap* conecta periódicamente con el servidor utilizando la clase `Update`.

La clase `Update` extiende la clase `Service` que, al igual que *Activity*, representa uno de los componentes básicos que pueden construir una aplicación Android. Un *Service*, tal y como se ha dicho en anteriores ocasiones, representa una tarea sin interfaz que se realiza en *background* sin conocimiento del usuario, y que puede formar parte del mismo proceso que la aplicación que lo lanza o de otro distinto.

Para actualizar los datos, la clase `Update` conecta de forma periódica con el servidor, utilizando la clase `HTTPConnection` y su método `connect()`, que se encarga de enviar al servidor la petición presente en su atributo `mDataOut` y escribir la respuesta en su otro atributo `mDataIn` (ver apartado 4.2.8). `Update` únicamente llama cada cierto tiempo a `connect()`, mientras que la clase `ContactMap`, también periódicamente, lee la respuesta recibida y escribe la petición que se ha de enviar en la próxima conexión que haga `Update`.

En la Figura 32 se muestra un diagrama de secuencia utilizando el estándar UML 2.0 [35] donde se exponen los pasos seguidos para realizar una conexión con el servidor. Nótese que la clase `ContactMap` utiliza las clases `XMLExchange` y `MyXMLHandler` para construir la petición y leer la respuesta, pero para simplificar dicho diagrama se han omitido esos pasos.

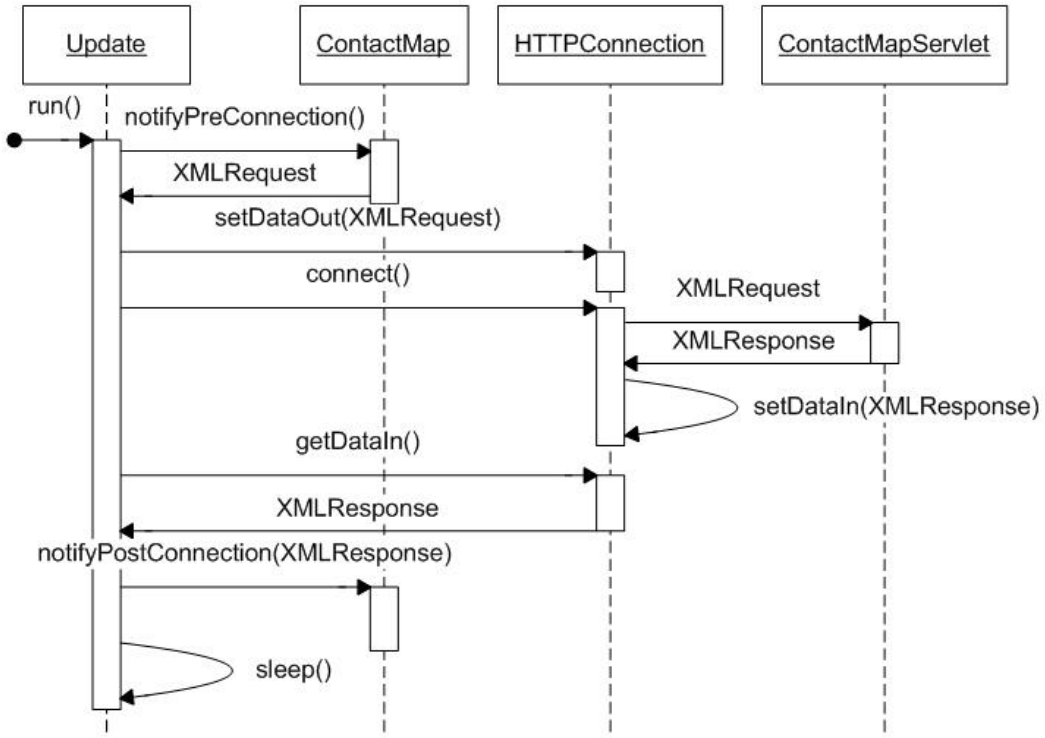


**Figura 32. Diagrama de secuencia para las conexiones con el servidor**

Lanzar el servicio es tan sencillo como llamar al método `startService()` de la clase `Activity`. Este hecho se produce en el método `onCreate()` de la clase `ContactMap`, justo cuando se ha comprobado que existe una conexión Wi-Fi disponible. El Código 61 muestra cómo se lanza el *Service* `Update`.

```
public class ContactMap extends MapActivity {

@Override
public void onCreate(Bundle savedInstanceState) {

      // (...)

      // Wi-Fi disponible
      if (wifiEnabled==true){

            // Crear un Intent asociado a la clase Update
            Intent UpdateService = new Intent(this, Update.class);
            // Lanzar Service
            this.startService(UpdateService);
```

```
      // Wi-Fi no disponible
      }else{
            // Lanzar aviso
            // Obtener localizaciones desde SQLite

      }
      // (...)
}

}
```

**Código 61. Lanzar el *Service* Update**

Al igual que ocurre cuando se lanza una nueva *Activity*, para lanzar un *Service* es necesario utiliza un objeto de la clase `Intent`. Este objeto se crea vinculado a la propia clase `Update`, que es el componente *Service* que se desea comenzar. Si existe Wi-Fi disponible, se lanza el servicio. Si no, se comunica tal situación al usuario y se actualizan los contactos utilizando la información guardada en la base de datos SQLite del propio dispositivo móvil (consultar apartados 4.2.7 y 4.2.12).

Generalmente, los elementos *Service* empleados en una aplicación Android repiten de forma cíclica una determinada tarea que debe ser simultánea a las otras actividades llevadas a cabo por la aplicación. Por ello, casi siempre un *Service* implementa un hilo o *thread* donde desarrolla alguna acción. `Update` lanza un hilo en su método `onCreate()`, tal y como se ve en el Código 62, que conecta con el servidor mediante `HTTPConnection` y se duerme un cierto tiempo mediante un `sleep()`.

```
public class Update extends Service{

// dirección del servidor
private String mServer = getResources().getString(R.string.server);
// temporizador de conexión
private int mTime = getResources().getInteger(R.integer.time_update);
// connexion HTTP
private HTTPConnection mHTTPConnection = new HTTPConnection(mServer);
// hilo que conectará periódicamente
public Thread mThread = new Thread(mTask);

@Override
public void onCreate() {

      super.onCreate();
      mThread.start();
}
```

```
Runnable mTask = new Runnable() {
      public void run(){

            while (true){

                  // Conectar con el servidor
                  mHTTPConnection.connect();
                  // Dormir el hilo 10 segundos
                  Thread.sleep(mTime);
            }
      }
};

}
```

**Código 62. Conexión periódica mediante la clase Update**

La clase `Update` cuenta además con atributo de nombre `mService` que contiene el nombre del servidor donde debe conectar *ContactMap*. Este es el único lugar donde se hace referencia a tal dirección, ya que en la construcción del objeto `HTTPConnection` se pasa como parámetro dicho valor y es el que utiliza el método `connect()` para lanzar su conexión HTTP. Para conocer la dirección del servidor, se utiliza el recurso de texto definido en el archivo "\res\values\strings.xml" de nombre "server". Mediante los métodos `getResources().getString()`, es posible obtener directamente una cadena de texto del recurso especificado.

Por otro lado, el atributo de `mTime` representa el intervalo de tiempo que debe esperar `Update` entre cada conexión. Este valor, representado en milisegundos, se obtiene también de los recursos externos, pero en este caso del fichero "\res\values\integers.xml", donde hay un elemento declarado con el nombre "update_timer".

### 4.2.19 Utilización de interfaces remotas mediante AIDL

En el apartado anterior se ha explicado como *ContactMap* actualiza de forma periódica su información con la del servidor, utilizando para ello un componente *Service* encarnado en la clase `Update`. Esta clase simplemente lanza el método `connect()` de `HTTPConnection`, que recoge en sus atributos tanto la petición que se ha de enviar como la respuesta que se recibe. La clase `ContactMap` es la que debe escribir y leer esos datos antes y después de la conexión, respectivamente.

Para actuar de forma coordinada, `Update` debe comunicar a la clase `ContactMap` cuándo se va a conectar, para que así pueda preparar la petición que se ha de enviar. Así mismo, `Update` también debe anunciar a `ContactMap` que la conexión ya se ha

producido y que existe disponible una respuesta del servidor que puede ser leída y procesada.

Estas comunicaciones han sido omitidas en el apartado 4.2.18 para centrar la atención del lector en el funcionamiento de un elemento *Service*, pero existen. Android permite cierto grado de comunicación entre una *Activity* y un *Service* lanzado por esta. Sin embargo, la comunicación en el otro sentido, tal y como se desea aquí, es algo más complejo y requiere el uso de interfaces remotas.

Una interfaz remota representa un mecanismo mediante el cual dos procesos separados pueden utilizar métodos declarados en el otro proceso. Cada uno de los extremos desconoce la implementación del otro, simplemente conoce a través de una interfaz qué métodos se ofrecen, pudiendo utilizarlos como si de cualquier otra clase local se tratase. Algunas tecnologías de interfaces remotas más utilizados son RMI de Java o CORBA. Android también cuenta con su propio mecanismo de interfaces remotas.

Antes de usar este tipo de comunicaciones en Android, es necesario realizar los siguientes pasos:

1. Declarar la interfaz remota mediante el lenguaje AIDL (*Android Interface Definition Language*). La declaración se realiza con una sintaxis muy básica e incluye los nombres de los métodos, los parámetros que éstos aceptan y los posibles valores devueltos. Con esta declaración se genera un resguardo o *stub* para ser utilizado por el otro extremo.

2. Implementar dicha interfaz.

3. Obtener el objeto remoto que permita el acceso a los métodos declarados.

**Declaración de las interfaces con AIDL y obtención de los *stubs***

El lenguaje AIDL de Android permite poder declarar interfaces remotas, generando además un resguardo para que el extremo que las va utilizar pueda conocer su naturaleza, aunque ignore su implementación. AIDL consiste en una simplísima sintaxis que se limita a anunciar métodos y tipos de datos básicos, además de tipos de cualquier clase perteneciente al paquete actual.

La clase `Update` necesita comunicar a `ContactMap` tanto el momento previo a la conexión con el servidor como el momento inmediatamente posterior, para que dicha clase puede escribir y leer los correspondientes documentos XML. Es decir, `ContactMap` necesita ofrecer dos métodos remotos a los que `Update` pueda acceder:

- `notifyPreConnection()`: donde se escribirá la petición.
- `notifyPostConnection()`: donde se leerá la respuesta.

Para que la clase `Update` pueda invocar estos métodos implementados en `ContactMap`, ha de tener un objeto que represente la interfaz remota de `ContactMap`. Se concluye, por tanto, que `Update` también ha de ofrecer otros métodos remotos que permitan a `ContactMap` pasarle un instancia de su interfaz remota con la que pueda invocar a sus métodos, es decir, hacer lo que se denomina *callback*. `Update` deberá implementar, pues, los siguientes métodos remotos:

- `register()`: guarda un objeto de la interfaz remota.
- `unRegister()`: elimina un objeto de la interfaz remota.

La pregunta que puede surgir ahora es que si `Update` necesita tener una instancia de la interfaz remota de `ContactMap` y registrarla para acceder a ella, ¿por qué a la inversa no se realiza el mismo proceso?; es decir, por qué `ContactMap` no necesita tener una instancia de la interfaz remota de `Update` y, por tanto, registrarla igualmente. La respuesta es que `ContactMap` sí necesita, como es lógico, una instancia de la interfaz remota de `Update` a la que pretende acceder, pero **no necesita registrarla** porque la obtiene automáticamente gracias al método `bindService()`. Esto se detalla más adelante. Como se ha dicho, es más simple comunicar una *Activity* con su *Service* que a la inversa.

Una vez conocidas las interfaces que se necesitan, el primer paso es declararlas formalmente a través de AIDL. El siguiente código muestra la declaración de la interfaz remota de `ContactMap`, llamada `IRemoteCallback`. En ella, el método `notifyPreConnection()` devuelve un valor *string*, que consistirá en la petición XML que se ha de enviar al servidor. Por su parte, el método `notifyPostConnection()` recibe como parámetro otro tipo *string* que representa la respuesta XML recibida desde el servidor.

```
package com.android.contactmap;


interface IRemoteCallback {

      String notifyPreConnection();
      void notifyPostConnection(String data);

}
```

**Código 63. Interfaz remota IRemoteCallback**

A continuación se muestra la interfaz remota implementada en la clase `Update`, que recibe el nombre de `IRemoteRegister`. Los métodos `register()` y `unRegister()` únicamente guardan y eliminan, respectivamente, una instancia de la interfaz remota `IRemoteCallback`.

```
package com.android.contactmap;
import com.android.contactmap.IRemoteCallback;

interface IRemoteRegister {

    void register(IRemoteCallback regService);
    void unRegister(IRemoteCallback regService);

}
```

**Código 64. Interfaz remota IRemoteRegister**

Estas declaraciones deben ser guardadas en ficheros con el mismo nombre que el de la interfaz y con la extensión ".aidl". Si se está trabajando con el *plug-in* de Eclipse, el simple hecho de colocarlas en el mismo proyecto donde van a ser utilizadas hará que se generen de forma automática dos ficheros Java, uno por cada interfaz, que representan el resguardo o *stub* que el otro extremo debe conocer para poder utilizarlas. En este caso, como ambos extremos `ContactMap` y `Update` están en el mismo proyecto, no hace falta trasladar ninguno de los resguardos.

### Implementación de las interfaces

Tras haber declarado las interfaces con el lenguaje AIDL, el siguiente paso es implementar los métodos declarados en ellas. La interfaz `IRemoteCallback` se corresponde con las llamadas que hará la clase `Update` a la clase `ContactMap` para avisarle tanto de que va a conectar con el servidor y necesita una petición, como que ha terminado dicha conexión y ya dispone de una respuesta.

Para implementar la interfaz `IRemoteCallback` en la clase `ContactMap` se declara un nuevo atributo de este tipo y se construyen los dos métodos anunciados. En el Código 65 se expone la implementación realizada.

```
public class ContactMap extends MapActivity {

// Lista de contactos
public ArrayList<Friend> mFriendList = new ArrayList<Friend>();
// Controlador del localizador GPS
private LocationManager mLocation = null;
// Gestor de intercambios con XML
private XMLExchange mXmlExchange = new XMLExchange();
```

```
private IRemoteCallback.Stub mCallback = new IRemoteCallback.Stub() {

      public String notifyPreConnection() {

            // Devolver petición XML
            return mXmlExchange.writeXMLUpdate(
                  getContentResolver(),
       (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE),
      mLocation.getLastKnownLocation(LocationManager.GPS_PROVIDER));

      }

      public void notifyPostConnection(String data) {

            // Procesar respuesta XML
            mXmlExchange.readXMLUpdate(
                  data, mFriendList, getContentResolver());

      }
};

}
```

**Código 65. Implementación de la interfaz IRemoteCallback**

Tal y como se ve en el Código 65, el método `notifyPreConnection()` utiliza la clase `XMLExchange` para construir un documento XML que represente una petición válida, donde se informe al servidor de la posición del usuario y se solicite las posiciones de los contactos. Por el contrario, el método `notifyPostConnection()` usa también dicha clase pero para leer la respuesta XML recibida del servidor, donde se encontrará la información de localización de los contactos.

Seguidamente, se muestra la implementación de la interfaz `IRemoteRegister` en la clase `Update`.

```
public class Update extends Service{

// Lista con interfaces remotas

private RemoteCallbackList<IRemoteCallback> mCallbacks =
new RemoteCallbackList<IRemoteCallback>();

private IRemoteRegister.Stub mRegister = new  IRemoteRegister.Stub(){

      public void register(IRemoteCallback interface) {
            // Guardar interfaz remota
            if (interface!= null) mCallbacks.register(interface);
      }
```

```
        public void unRegister(IRemoteCallback interface) {
                // Eliminar interfaz remota
                if (interface!= null) mCallbacks.unregister(interface);
        }
};
}
```

**Código 66. Implementación de la interfaz IRemoteRegister**

Antes de implementar la interfaz, se declara una instancia de la clase `RemoteCallbackList`, perteneciente al paquete `android.os`. Esta clase permite manejar más fácilmente un conjunto de interfaces remotas especialmente pensadas para hacer *callback* desde un *Service*. En el caso aquí explicado, a priori solamente habrá un único elemento en esta lista: la instancia de la interfaz `IRemoteCallback`.

Los métodos remotos `register()` y `unRegister()` simplemente añaden o eliminan una interfaz remota a la lista. De esta forma, una vez está incluida en dicha lista, se puede acceder sin mayor complicación a los métodos remotos ofrecidos por esa interfaz.

### Utilización de las interfaces

Llegado este punto, las interfaces remotas ya han sido declaradas mediante AIDL, se ha generado automáticamente su *stub* o resguardo y además se han implementado los métodos que cada interfaz ofrece: la interfaz `IRemoteCallback` está implementada dentro de la clase `ContactMap`, y la interfaz `IRemoteRegister` lo está en la clase `Update`. Ahora es necesario proporcionar a cada clase una instancia de la interfaz remota que necesita, esto es, `ContactMap` precisa de una instancia de `IRemoteRegister` y `Update` requiere una instancia de `IRemoteCallback`.

Para ofrecer a `ContactMap` una instancia de `IRemoteRegister` es necesario utilizar el método `bindService()` de la clase `Context`. . La clase `Service` tiene también un ciclo de vida bien definido, es decir, cuenta con métodos como `onCreate()`, `onStart()`, `onDestroy()`, etc. Cuando se lanza un elemento *Service*, por defecto este es independiente al ciclo de vida de la *Activity* que lo ha lanzado, lo que implica que el *Service* seguirá ejecutándose aunque la *Activity* que lo ha lanzado muera, y solamente finalizará cuando la *Activity* lo haga de forma explícita.

En esta situación no se desea que el *Service* sea independiente. Por un lado, interesa vincular el ciclo de vida de la clase `ContactMap` y la clase `Update` para que cuando la aplicación *ContactMap* finalice, la conexión periódica con el servidor también lo haga; por otra, la utilización de `bindService()` va a proporcionar a la clase `ContactMap` una instancia de la interfaz remota que necesita, `IRemoteRegister`.

En el siguiente código, el Código 67, se muestra al lector cómo se utiliza `bindService()` para poder lanzar correctamente el servicio `Update` de la forma que aquí interesa.

```
public class ContactMap extends MapActivity {

      @Override
      public void onCreate(Bundle savedInstanceState) {

      // (...)

      // Wi-Fi disponible
      if (wifiEnabled==true){

            // Vincular el Service y la Activity actual
            bindService(new Intent(IRemoteRegister.class.getName()),
                              mConnection, Context.BIND_AUTO_CREATE);

            // Crear un Intent asociado a la clase Update
            Intent UpdateService = new Intent(this, Update.class);

            // Lanzar Service
            this.startService(UpdateService);

      // Wi-Fi no disponible
      }else{

            // Lanzar aviso
            // Obtener localizaciones desde SQLite

      }

      // (...)
}
}
```

**Código 67. Lanzar el *Service* Update usando interfaces remotas**

Este código es íntimamente similar al mostrado en el Código 61, salvo que antes de lanzar el servicio con `startService()` se utiliza el método `bindServicer()`, vinculando así la clase `ContactMap` y la clase `Update`. Se pueden apreciar tres parámetros en este método:

- Un objeto de la clase `Intent`.
- Un objeto de la clase `ServiceConnection`, llamado aquí `mConnection`.
- Un flag de valor `Context.BIND_AUTO_CREATE` para la creación del elemento *Service*.

Con respecto al primer parámetro de `bindServicer()`, el lector ya sabrá que un *Intent* siempre representa una acción que se desea realizar. Al crear un *Intent*, o bien se especifica qué acción se desea llevar a cabo utilizando alguna de las constantes que ofrece Android (ver los apartados 4.2.15 ó 4.2.16) o bien se especifica directamente la clase que se quiere lanzar (repásense los casos descritos en los apartados 4.2.14 ó 4.2.17). Aquí se opta por el segundo caso. Lo que se está diciendo a través del *Intent* es que se desea ejecutar la interfaz `IRemoteRegister`. Ya que la clase `IRemoteRegister` presente en el proyecto no es una clase ejecutable, sino un simple *stub* o resguardo, debe existir algún otro elemento que indique explícitamente que puede hacerse cargo de *Intents* de ese tipo. Dicho esto, se comprederá ahora la necesidad de que en el fichero "AndroidManifest.xml", el *Service* ya declarado `Update` anuncie ahora que puede atender este tipo de *Intents*:

```
<service android:name=".Update">
      <intent-filter>
      <action android:name="com.android.contactmap.IRemoteRegister" />
      </intent-filter>
</service>
```

**Código 68. Declaración en el manifiesto de los *Intents* para el *Service* Update**

Con la etiqueta `<intent-filter>` cualquier componente de una aplicación Android puede anunciar que es capaz de hacerse cargo de unos determinados *Intents*. De hecho, esta etiqueta es consultada por Android una por una en las aplicaciones instaladas cuando alguien lanza un *Intent*, a fin de encontrar a la aplicación más adecuada para hacerse cargo. Así pues, cuando alguien lance un *Intent* asociado a `IRemoteRegister`, la clase `Update` se hará cargo de ello.

El segundo parámetro de `bindService()` es un objeto de la clase `ServiceConnection`. Esta clase pertenece al paquete `android.content` y es en realidad una interfaz que debe ser implementada por el desarrollador. Ya que `bindService()` permite vincular al *Service* y a la *Activity*, debe existir un objeto que monitorice cuál es el estado del *Service*. Este es el cometido del objeto `ServiceConnection`. En concreto, ofrece dos métodos: `onServiceConnected()` y `onServiceDesconnected()`. Como se podrá intuir, el primero es llamado cuando el *Service* y la *Activity* se conectan, mientras que el segundo lo hace cuando se desconectan.

La llamada a `bindService()` invoca en el lado del componente *Service* un método denominado `onBind()`. La finalidad de este método está enfocada casi en exclusiva a la utilización con interfaces remotas, ya que su objetivo es devolver a la *Activity* un objeto mediante el cual pueda comunicarse con el *Service*. Este objeto ha de ser una instancia de la interfaz `IRemoteRegister`, tal y como se ve en el siguiente código.

```
public class Update extends Service{

private IRemoteRegister.Stub mRegister = new  IRemoteRegister.Stub(){

      // Aquí se implementan los métodos register() y unRegister()

}

@Override
public IBinder onBind(Intent intent) {

      return mRegister;
}

}
```

**Código 69. Método onBind() de la clase Update**

Así pues, la clase `ContactMap`, tras invocar al método `bindService()` y este al método `onBind()`, obtiene una instancia de la interfaz `IRemoteRegister`. Este objeto se recibe a través del método `onServiceConnected()`, que forma parte de la interfaz de `ServiceConnection` y debe ser implementado por el desarrollador. La finalidad de tener una instacia de `IRemoteRegister`, como se recordará, es permitir a la clase `ContactMap` registrar en la clase `Update` una instancia de `IRemoteCallback`, de forma que el círculo de comunicación ya quede cerrado.

```
public class ContactMap extends MapActivity {

// Interfaz remota de la clase Update
private IRemoteRegister mService = null;

// Interfaz remota de la clase ContactMap
private IRemoteCallback.Stub mCallback = new IRemoteCallback.Stub() {

      // Aquí se implementan los método notifyPreConnection() y
      // notifyPostConnection()

};

private ServiceConnection mConnection = new ServiceConnection() {

public void onServiceConnected(ComponentName className,
IBinder service) {

      // Obtener instacia gracias al stub
      mService = IRemoteRegister.Stub.asInterface(service);
```

```
      // Hacer llamada remota: registrar instancia de IRemoteCallback
      mService.register(mCallback);
}

public void onServiceDisconnected(ComponentName className) {
      mService = null;
}

};
}
```

**Código 70. Implementación de la interfaz ServiceConnection**

Cuando se llama a `bindService()`, los métodos `onBind()` y `onServiceConnected()` se invocan. El primero devuelve una instancia de la interfaz `IRemoteRegister` y el segundo permite recogerla y transformarla correctamente. Esta transformación se realiza gracias al *stub* o resguardo de `IRemoteRegister`. Desde el momento que ya se tiene una instacia correcta de `IRemoteRegister`, encarnada en el atributo `mService`, ya se pueden utilizar los métodos remotos ofrecidos. Por ello, `ContactMap` registra de inmediato una instancia de su interfaz remota `IRemoteCallback`, permitiendo desde ese instante que la clase `Update` pueda comunicarse con `ContactMap`.

La clase `Update` tiene que llamar al método `notifyPreConnection()` justo antes de realizar una conexión, para que `ContactMap` escriba la petición a enviar, e invocar a `notifyPosConnection()` justo depués para que `ContactMap` lea la respuesta. Todo este proceso se realiza en el hilo que la clase `Update` crea para realizar las conexiones de forma periódica. Véase el Código 71.

```
public class Update extends Service{
// connexion HTTP
private HTTPConnection mHTTPConnection = new HTTPConnection(mServer);
// hilo que conectará periódicamente
public Thread mThread = new Thread(mTask);

Runnable mTask = new Runnable() {
public void run()
{

      // Mientras dure el Service
      while (true){

            // Activar mCallbacks
            mCallbacks.beginBroadcast();
            // Notificar a la clase ContactMap
            mHTTPConnection.setDataOut(
            mCallbacks.getBroadcastItem(0).notifyPreConnection());
```

```
            // Conectar con el servidor
            mHTTPConnection.connect();

            // Notificar a la clase ContactMap
            mCallbacks.getBroadcastItem(0).notifyPostConnection(
            mHTTPConnection.getDataIn());
            // Desactivar mCallbacks
            mCallbacks.finishBroadcast();

            // Dormir el hilo
            Thread.sleep(mTime);
      }
}
};

}
```

**Código 71. Notificaciones desde la clase Update a la clase ContactMap**

Este código es muy similar al mostrado en el Código 62, exceptuando que aquí se incluyen las notificaciones enviadas a `ContactMap` antes y después de la conexión con el servidor. La clase `RemoteCallbackList`, que representa una lista con los objetos remotos disponibles para hacer *callback*, requiere ser activado antes de las llamadas remotas y desactivado después. Este requisito se cumple con los métodos `beginBroadcast()` y `finishBroadcast()`, respectivamente. Las notificaciones se envían justo antes de la conexión, obteniendo de ella la petición a enviar, así como después de la misma, donde se envía por parámetro la respuesta recibida.

### 4.2.20 Manifiesto final

En este apartado se ofrece al lector el aspecto final del fichero "AndroidManifest.xml" utilizado en la aplicación *ContactMap*.

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3. package="com.android.contactmap"
4. android:versionCode="1"
5. android:versionName="1.0.0">
6.
7. <uses-permission android:name="android.permission.READ_PHONE_STATE" />
8. <uses-permission android:name="android.permission.INTERNET" />
9. <uses-permission android:name="android.permission.READ_CONTACTS" />
10. <uses-permission android:name="android.permission.CALL_PHONE" />
11. <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
    />
12. <uses-permission android:name="android.permission.SEND_SMS" />
13. <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
14.
15.
16.
```

```
17.
18. <application android:icon="@drawable/icon"
    android:label="@string/app_name">
19.
20. <uses-library android:name="com.google.android.maps" />
21.
22. <activity android:name=".ContactMap" android:label="@string/app_name">
23.       <intent-filter>
24.       <action android:name="android.intent.action.MAIN" />
25.       <category android:name="android.intent.category.LAUNCHER" />
26.       </intent-filter>
27. </activity>
28.
29. <activity android:name=".FriendListViewer"></activity>
30.
31. <activity android:name=".SMSWriter"></activity>
32.
33. <service android:name=".Update">
34.       <intent-filter>
35.       <action android:name="com.android.contactmap.IRemoteRegister" />
36.       </intent-filter>
37. </service>
38.
39. </application>
40.
41. </manifest>
```

**Código 72. Manifiesto final de *ContactMap***

Las principales declaraciones de este manifiesto son las siguientes:

- Líneas 7-13: permisos para la aplicación. Por orden de aparición, están los permisos para acceder a la información del dispositivo móvil, acceder a Internet, leer los contactos del dispositivo, realizar llamadas, control del elemento de localización, enviar SMS y control de la Wi-Fi.
- Líneas 18-39: declaración de los componentes que forman la aplicación.
- Línea 20: uso de la librería de Google Maps.
- Líneas 22-27: declaración de la *Activity* principal, la clase `ContactMap`.
- Línea 24: la clase `ContactMap` es la clase principal.
- Línea 25: la clase `ContactMap` debe ejecutarse nada más ejecutar la aplicación.
- Línea 29: declaración de la *Activity* `FriendListViewer`.
- Línea 31: declaración de la *Activity* `SMSWriter`.
- Líneas 33-37: declaración del *Service* `Update`.
- Línea 35: interfaz remota ofrecida por el *Service* `Update`.

# 5 HISTORIA DEL PROYECTO

La elección de Android como tema para este proyecto final de carrera tuvo lugar en primavera de 2008, después de valorar las diferentes alternativas que hasta entonces se habían encontrado. En aquel momento el autor del presente proyecto se encontraba en mitad del segundo cuatrimestre del 5º curso de Ingeniería en Informática; desde hacía meses atrás se había estado consultando con distintos profesores posibles temas para desarrollar, dada la cercana finalización de todas las asignaturas. Durante estas consultas, acudió principalmente a aquellos profesores cuyas asignaturas habían resultado más interesantes o sencillamente a aquellos con los que mejor relación se había entablado.

Finalmente, la elección de un tema como Android se basó en que representaba un tema, al menos para el autor de este proyecto, sumamente interesante por varias razones: se trataba de un sistema muy novedoso, lanzado por una de las empresas fetiche de Internet como es Google, y que tenía como objetivo el prometedor mercado de los dispositivos móviles.

El proyecto comenzó, por tanto, en abril de 2008 y fue finalizado en enero de 2009, es decir, tuvo un periodo de duración aproximado de 10 meses. Durante este periodo de tiempo el ritmo de trabajo no fue uniforme, ya que tuvo que ser compaginado con otras actividades académicas. El tiempo de trabajo real invertido en todo su desarrollo ha sido de **un total de 652 horas**.

Desde el primer día de trabajo se ha llevado una rigurosa cuenta de las horas invertidas en cada tarea. En total, se puede dividir el trabajo en 3 actividades básicas:

- **Estudio de Android**: comprende todas las tareas relacionadas con el estudio y comprensión del sistema Android: la lectura de la documentación, la instalación del SDK, el estudio de las distintas API, análisis de la arquitectura, su funcionamiento interno, tutoriales, estudio de aplicaciones sencillas de ejemplo, etc. El tiempo total aquí empleado fue de **117 horas**.

- **Desarrollo de la aplicación**: incluye todas aquellas tareas vinculadas al desarrollo completo de una aplicación como Android. Fundamentalmente, el análisis y su diseño, la implementación y las pruebas. La duración de esta actividad fue de **349 horas**.

- **Documentación del proyecto**: la documentación abarca las tareas relacionadas con la redacción completa de la presente memoria, donde se incluye la descripción de Android y la aplicación desarrollada. Esta actividad tuvo una duración de **186 horas**.

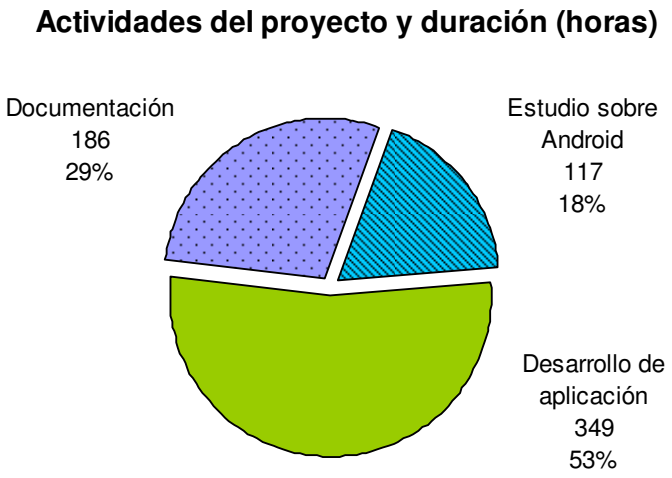


**Figura 33. Actividades del proyecto y su duración en horas**

El tiempo empleado en al realización de las distintas actividades a lo largo de los meses de duración no ha sido, como se ha comentado anteriormente, uniforme, sino que ha estado sujetado a otros aspectos académicos. A continuación, se una breve descripción del historial del proyecto mes a mes. En la Tabla 3 puede observarse el tiempo dedicado a cada actividad según el mes de desarrollo.

- **Abril, Mayo, Junio**: a principios del mes de abril el proyecto fue asignado de forma oficial. La única actividad desarrollada durante abril y mayo fue una toma de contacto con el nuevo sistema Android, recavando y conociendo sus características más básicas. Así mismo, se descargó el SDK, se instaló el *plug-in* para Eclipse y se hicieron los tutoriales disponibles en la documentación oficial. Durante el mes de junio se comenzó a redactar las primeras páginas de la documentación, explicando las características de Android aprendidas hasta entonces. Es importante señalar que estos meses coinciden con la parte más dura del segundo cuatrimestre de 5º curso, además del periodo de exámenes.

- **Julio, Agosto, primera quincena de Septiembre**: durante estos meses correspondientes al periodo de verano el proyecto fue temporalmente aparcado, debido a que en el mes de septiembre se tenía pendiente la última asignatura de la carrera, una asignatura troncal, que incluía tanto examen como práctica.

- **Segunda quincena de Septiembre**: una vez superado el periodo estival y aprobada la asignatura pendiente, el proyecto fue de nuevo retomado. Durante los días restantes del mes de septiembre se hizo un repaso de los conocimientos hasta ahora obtenidos, se estudiaron algunas aplicaciones ya disponibles para Android y se redactó nueva documentación, en concreto el estado del arte del proyecto.

- **Octubre**: este mes constituye el primero que se pudo dedicar en exclusiva al proyecto, por lo que las horas dedicas aumentan considerablemente. En primer lugar, se terminó definitivamente la fase de estudio de Android, pasando a continuación a redactar toda la parte de la documentación que consistiera en describir los pormenores de este sistema, correspondiente básicamente al tercer capítulo. Así mismo, los últimos días de este mes se comenzó a pensar y a diseñar la aplicación, realizando la implementación de sus componentes más básicos y generales.

- **Noviembre**: durante todo el mes de noviembre la única actividad fue el desarrollo de la aplicación: instalación del lado servidor, conexiones HTTP, gestión de intercambios XML, GPS, Wi-Fi y uso de mapas con Google Maps, principalmente.

- **Diciembre**: la primera quincena de diciembre estuvo dedicada también en exclusiva al desarrollo de la aplicación (menú de opciones, envío de SMS, llamadas telefónicas y envío de correos electrónicos, actualización con el servidor, interfaces remotas), y también la corrección de errores y pruebas. La redacción de la documentación correspondiente a describir el desarrollo y funcionamiento de la aplicación (es decir, el cuarto capítulo) tuvo lugar la segunda quincena de este mes.

- **Enero:** el mes de enero, último en el desarrollo del proyecto, abarcó tareas únicamente relacionadas con redactar aquellas últimas partes de la documentación, como las conclusiones, trabajos futuros, planificación, etc.

| | Estudio | Desarrollo | Documentación | Total mes |
|---|---|---|---|---|
| *Abril* | 26 | 0 | 0 | **26** |
| *Mayo* | 32 | 0 | 0 | **32** |
| *Junio* | 0 | 0 | 11 | **11** |
| *Julio* | 0 | 0 | 0 | **0** |
| *Agosto* | 0 | 0 | 0 | **0** |
| *Septiembre* | 36 | 0 | 30 | **66** |
| *Octubre* | 23 | 59 | 62 | **144** |
| *Noviembre* | 0 | 168 | 0 | **168** |
| *Diciembre* | 0 | 122 | 63 | **185** |
| *Enero* | 0 | 0 | 20 | **20** |
| | **117** | **349** | **186** | **652** |

**Tabla 3. Tiempo dedicado cada mes a cada actividad (horas)**

En la siguiente figura, la Figura 34, se ofrece al lector una gráfica en la que puede comprobar el número de horas dedicas a cada una de las actividades del proyecto según el mes de desarrollo del mismo:

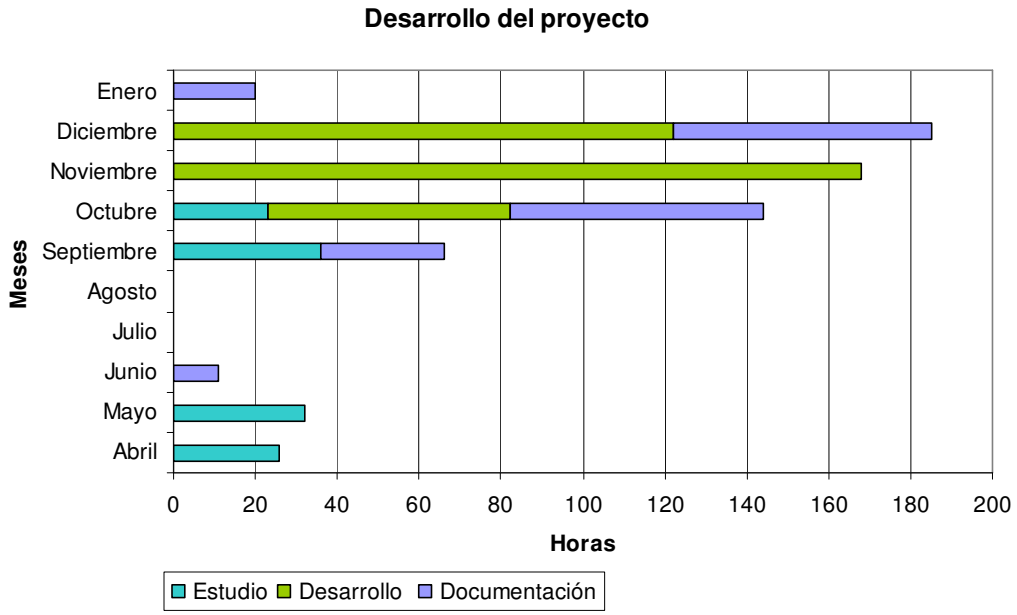


**Figura 34. Desarrollo del proyecto por meses y actividades**

A lo largo de las distintas actividades del proyecto, como es natural, se fueron encontrando algunas dificultades o contratiempos no previstos. Por ejemplo, en septiembre de 2008 Google publicó la primera versión estable de Android, la 1.0, que es la que lleva actualmente el único terminal comercializado. Esta versión cambió ligeramente algunos paquetes y sus clases, pero sobre todo la llamada a ciertos métodos. Esto provocó la revisión obligada de parte de la información que se había recavado antes del mes de septiembre sobre el funcionamiento de Android; además, tuvo un segundo efecto y fue que muchas de las aplicaciones publicadas libremente por desarrolladores ya no podían ser ejecutadas en el nuevo entorno y, por lo tanto, ya no pudieron utilizarse para observar in situ algunas de las posibilidades ofrecidas por Android.

En este misma línea, y tal y como se comenta en el siguiente capítulo, a pesar de la concienzuda documentación que Google ofrece a la comunidad a través de la web oficial de Android, se echan en falta ejemplos prácticos, aplicaciones sencillas o tutoriales (más allá del ejemplo del Notepad ya existente) que permitan al desarrollador comprobar de forma práctica cómo deben utilizarse las diferentes clases para realizar según qué acciones. La utilización de componentes como el GPS, o la conexión Wi-Fi, los pasos son muy sencillos y elementales, pero en otros como el uso de interfaces remotas con AIDL se encontraron verdaderas dificultades para ponerlas en práctica.

En cuanto al presupuesto necesario para el desarrollo de este proyecto, va a tomarse como necesaria la presencia de dos trabajadores:

- **Analista**: encargado de estudiar Android, diseñar la aplicación y redactar la documentación más relevante, representada por dos tercios de la presente memoria.

- **Programador**: encargado de la implementación de la aplicación y la documentación relativa a la misma (una tercera parte de la memoria).

Además, es necesario tener en cuenta el hardware necesario para poder realizar todo el proyecto. En este caso se ha utilizado únicamente una máquina de desarrollo, pero es de suponer que sería necesario además disponer de un terminal con Android, liberado para poder usarlo actualmente en España:

- **Ordenador portátil Asus X59SL**, con procesador Intel Core Duo T5800 a 2 GHz, memoria RAM DDR de 4 GB, disco duro de 320 GB de capacidad, entre otros.

- **Dispositivo HTC G1**, con procesador Qualcomm MSM7201A a 528 MHz, memoria RAM de 192 MB, pantalla LCD de 3.2 pulgadas, GSM, GPRS, Bluetooth, GPS y Wi-Fi, entre otros componentes.

El costo de cada profesional, según la media existente en España para el año 2008 [40], es de 18,23 euros\hora y 15,63 euros\hora, respectivamente. Por ello, tal y como se ve en la Tabla 4 donde se expone el coste tanto del personal como de los recursos hardware, el coste final del proyecto es de **11905,94 euros**.

| **PERSONAL** | | | | |
|---|---|---|---|---|
| | **Estudio (h)** | **Desarrollo (h)** | **Documentación (h)** | **Euros** |
| **Analista** | 117 | 0 | 124 | 4393,43 |
| **Programador** | 0 | 349 | 62 | 6423,93 |
| **HARDWARE** | | | | |
| | **Euros** | | | |
| **Asus X59SL** | 668,59 [41] | | | |
| **HTC G1** | 419,99 [42] | | | |
| **TOTAL** | | | | 11905,94 |

**Tabla 4. Coste del proyecto**

# 6 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se hace un repaso global al proyecto, presentado las conclusiones finales en comparación con los objetivos marcados inicialmente y los posibles trabajos futuros.

## *6.1 Conclusiones finales*

Una vez concluidas las principales tareas que forman este proyecto fin de carrera, es el momento en el que se puede hacer balance y crítica de los resultados obtenidos. Así pues, repasando los objetivos inicialmente marcados pueden sacarse las siguientes conclusiones:

- **Conocer las principales características de Android.**

  A lo largo de todo el presente proyecto se ha conseguido obtener un conocimiento amplio de este nuevo sistema operativo. Su arquitectura, sus componentes y características, así como el funcionamiento y posibilidades ofrecidas por un sistema como Android se han ido conociendo gracias principalmente a la extensa y, en general, completa documentación que Google ha puesto a disposición de los desarrolladores. Especialmente en las primeras fases, esta documentación es útil y fácil de asimilar, lo que permite acercarse poco a poco a las formas y la tecnología de esta nueva plataforma.

  Tras comenzar a estudiar las características de Android, pueden observarse algunos aspectos que, si bien no siempre resultan una ventaja frente a sus competidores, sí son interesantes y pueden repercutir positivamente en su elección como plataforma. Por ejemplo:

  - A pesar de que Google evita usar el término demasiado, el hecho de utilizar un lenguaje tan popular como **Java** ayuda a que cualquier programador mínimamente experimentado pueda comenzar a programar sus aplicaciones sin mayor complicación, además de animar a los que ya estén muy familiarizados. Incluye, además, las API más importantes de este lenguaje como `java.util`, `java.io` o `java.net`.

  - La **licencia Apache** permite a todo el mundo poder estudiar, modificar y distribuir el sistema Android, a la vez que da opción al desarrollo privado mediante la publicación comercial de aplicaciones. Cada desarrollador puede decidir cómo quiere distribuir su propio trabajo.

  - Como ya es sabido, Android divide todas sus aplicaciones en **componentes o bloques básicos** que, combinados, constituyen el programa final. Así, tenemos bloques visibles para el usuario mediante interfaces (*Activity*),

bloques que se ejecutan en *background* fuera de su conocimiento (*Service*), bloques a la escucha de determinados eventos (*Broadcast Receiver*) y bloques que ofrecen contenidos a otras aplicaciones (*Content Provider*s). Esta filosofía es original y ayuda a modularizar funcionalmente las aplicaciones.

- La delegación de acciones en otras aplicaciones mediante ***Intents*** es otro de los aspectos más innovadores ofrecidos por Android. Mediante un *Intent*, la aplicación simplemente expresa lo que desea hacer y es el sistema el encargado de buscar la aplicación más adecuada (para llamar, mandar un correo electrónico, abrir una página web, etc.). Así mismo, las aplicaciones pueden anunciar a las demás que están preparadas para poder atender determinados tipos de *Intents*.

- La **construcción de interfaces** de usuario ha sido un aspecto muy cuidado en Android, no sólo por la amplia colección de elementos y diseños incorporados, sino por la posibilidad de ser definidas tanto en el código fuente como mediante documentos XML externos.

- El acceso a los **recursos del dispositivo**, como GPS, Wi-Fi, etc., se convierte en una tarea fácil y simple gracias a las API que Android ofrece en su SDK. Claramente se percibe que Android ha sido creado pensando en los dispositivos móviles más avanzados, por lo que cuando en pocos años estos sean mayoría será cuando Android puede demostrar toda su capacidad.

- La declaración y uso de **recursos externos**, tales como imágenes, cadenas de texto, valores numéricos, o incluso diferentes modelos de interfaz de usuario y de diseños es cómoda y fácil de realizar, dando un aspecto realmente elegante a la programación de aplicaciones en Android.

Además de la documentación ofrecida por Google, otro de los elementos utilizados para indagar en los entresijos de Android han sido los foros, blogs y demás publicaciones en Internet de la todavía pequeña comunidad de desarrolladores en este sistema. Es de esperar que, con el tiempo, aparezcan muchos más de estos elementos que combinados con la documentación oficial ayuden a conseguir una mejor y más rápida visión de Android.

- **Estudiar el entorno de desarrollo de Android.**

Gran parte de la documentación de Android consiste en un desglose pormenorizado de sus paquetes, clases e interfaces. Estas secciones, junto a las API Demos, resultan sumamente útiles e imprescindibles para poder conocer realmente las capacidades de Android y saber cuál es el reparto de responsabilidades. Las principales API de Android y sus ejemplos fueron consultados y estudiados para ver las posibilidades del sistema y poder comenzar a perfilar la naturaleza de la futura aplicación a desarrollar.

Por otro lado, el SDK de Android viene acompañado de un *plug-in* para Eclipse que facilita enormemente la tarea de programación en este sistema. Así, pueden crearse proyectos completos para Android, incluyendo el manifiesto y la declaración de recursos externos, sin salir del entorno de desarrollo de Eclipse. Además, en la ejecución se utiliza el emulador adjunto al SDK, otro complemento que incluye valiosas opciones para la depuración y construcción de las aplicaciones, como redireccionamiento de puertos entre emulador y máquina de desarrollo o el establecimiento de rutas GPS simuladas. Muchas de estas herramientas han sido utilizadas y en parte documentadas en el presente proyecto.

- **Desarrollar una aplicación completa para Android.**

  Después de haber cumplido los objetivos anteriores, y conocer las características de Android, sus API y el entorno de desarrollo existente, se procedió a desarrollar una aplicación completa para este sistema.

  *ContactMap* permite localizar los contactos almacenados en el dispositivo utilizando para ello mapas obtenidos desde Google Maps. El objetivo principal de este desarrollo no es tanto la utilidad de la aplicación final, sino el empleo y explicación de las principales características que Android ofrece frente a otros sistemas, además de orientar al lector en el desarrollo de futuras aplicaciones.

  En *ContactMap* se utilizan diferentes componente básicos, como *Activity* o *Service*; también se controlan elementos de dispositivo móvil como el GPS, básico para conocer la propia posición, o la Wi-Fi, para poder intercambiar información de localización con otros usuarios; además, se componen varias interfaces de usuario y se explica cómo lanzar *Intents* para enviar SMS, correos electrónicos o llamadas telefónicas; igualmente, *ContactMap* contempla el uso de interfaces remotas, conexiones por HTTP, así como la declaración completa de un manifiesto y una jerarquía de recursos externos.

  Todo el diseño e implementación de la aplicación ha sido documentado convenientemente en esta memoria con el fin de ayudar al lector a comprender el funcionamiento y construcción de aplicaciones para Android, representando así un caso práctico donde se aplican las características explicadas en capítulos previos.

## Críticas y dificultades

Los objetivos marcados para este proyecto han sido, por lo tanto, satisfechos. El balance obtenido del mismo es muy positivo, ya que se ha conocido de forma más o menos amplia un nuevo y prometedor sistema operativo para dispositivos móviles, aparatos que en pocos años formarán parte (si no lo son ya) de nuestra vida cotidiana en todas sus facetas. Sin embargo, aunque se han narrado las ventajas y cualidades que parece

ofrecer Android, no se quiere dejar pasar la oportunidad de realizar algunas críticas y exponer algunas dificultades que se han encontrado en este proyecto:

- La documentación ofrecida por Google ha sido, como se ha comentado anteriormente, la principal fuente de conocimiento sobre Android que aquí se ha utilizado. La parte que más abarca dicha documentación es la referente a la exposición de paquetes y clases de Android, donde sí que se ha echado en falta un **mayor contenido práctico**. Cada clase, método y parámetro es descrito de forma muy meticulosa y precisa, pero en algunos casos resultan explicaciones insuficientes cuando se quieren poner en práctica. Es entonces cuando han entrado en juego las comunidades de desarrollo de Android, con sus foros, blogs y demás publicaciones.

  Android cuenta con varios ejemplos de implementaciones reunidas en las API Demos, pero estas también carecen a veces de explicaciones claras y es necesario saber muy bien qué es lo que se busca para encontrarlo. En el caso de la aplicación *ContactMap*, se encontraron algunas dificultades para conocer cómo realizar una conexión HTTP utilizando las clases específicas de Android, cómo situar elementos sobre el mapa que se desplacen junto a este y, sobre todo, en la utilización de interfaces remotas. Todos estos aspectos se resolvieron mucho más fácilmente consultando el material que ponen a disposición las distintas comunidades de desarrolladores de Android surgidas en los últimos meses (para más información de comunidades sobre Android, consultar referencias [43], [44], [45] y [46]).

- Muy relacionado con el anterior punto, una de las dificultades encontradas más recurrentemente fue que, aunque existan **foros y comunidades** de intercambio de desarrolladores, estos son aún muy escasos. Dada la diferencia que en ocasiones existe entre el SDK 1.0 estable, surgido en septiembre de 2008, y los SDK anteriores en estado beta, quedaba invalidado además gran parte del material que en estas comunidades existía. Aún así, es de esperar que según se implante Android, surgirán en Internet más grupos y ejemplos de este tipo.

- Hay algunos aspectos de Android que, de momento, resultan un tanto complicados de utilizar. El primero de ellos son las **interfaces remotas**, utilizadas en la aplicación *ContactMap*. Sumado al hecho de que su explicación es bastante limitada en la documentación oficial, esto puede terminar de confundir al desarrollador.

  Otro aspecto no muy claro es el acceso a algunos proveedores de datos o *Content Provider* incluidos en el sistema, como los correos electrónicos de los contactos, o las imágenes asociadas. Todos estos datos están contenidos en tablas distintas representadas por varias clases, de forma que en muchos casos es difícil encontrar en la documentación dónde se encuentra el campo buscado y cómo se accede a esa tabla. Además, de momento las consultas no permiten el cruce de tablas.

- Resulta cuando menos sorprendente el hecho de que el SDK 1.0, la primera versión estable y comercializada de Android, **limite el uso del Bluetooth** a conexiones de elementos inalámbricos como auriculares o manos libres, imposibilitando el intercambio de datos. Esta peculiaridad, sumada al hecho de que con otro tipo de conexiones (Wi-Fi, GPRS, UMTS) no se pueda **elegir libremente el protocolo** a utilizar en un intercambio de datos (ver apartado 4.2.7), hace pensar que Android presenta todavía algunas características propias de una versión beta más que de un versión estable, probablemente debido al ansia de publicar antes de tiempo una versión comercializable.

- Por último, hubiera sido interesante **disponer de algún dispositivo** móvil donde probar las características de Android. En el momento de esta redacción, únicamente existe un modelo de Android a la venta, pero es lógico pensar que con el tiempo cualquier desarrollador podrá pobrar en su propio dispositivo las características de este nuevo sistema.

## 6.2 Trabajos futuros

Como líneas futuras de trabajo, se proponen los siguientes puntos:

- Completar la documentación de los principales paquetes y clases de Android con ejemplos específicos que ilustren claramente su uso y finalidad. De esta forma, se podría intentar compensar la falta de este tipo de material de la que adolece la documentación oficial de Google. La idea sería hacer algo similar a la explicación de la aplicación *ContactMap*, pero específica para los paquetes y clases más relevantes del sistema.

- Ampliar las funcionalidades ofrecidas por la aplicación *ContactMap*; por ejemplo, permitir al usuario delimitar una zona del mapa, de forma que la aplicación lance un aviso cuando un contacto determinado entre en dicha zona. También sería interesante introducir conceptos de seguridad y privacidad: identificación a la hora de conectar con el servidor, obligar a que los contactos den su permiso explícito a los usuarios de *ContactMap* para ser localizados (algo similar a lo ocurrido en aplicaciones de mensajería instantánea) o poder activar y desactivar voluntariamente la información de localización del usuario (lo que se podría llamar un "modo invisible")

- Probar la aplicación *ContactMap* en un dispositivo móvil real con Android. Actualmente, sólo existe un terminal a la venta, pero con la paulatina implantación de Android se dispondrán de varios modelos distintos y probablemente de precios más asequibles. Además, Google ha puesto en marcha una iniciativa mediante la cual ofrece a los desarrolladores la oportunidad de adquirir un modelo HTC G1 para realizar pruebas con sus aplicaciones [47].

# 7 ANEXO A: MANUAL DE INSTALACIÓN DE CONTACTMAP

En este capítulo se ofrece un manual de instalación de la aplicación *ContactMap*. Como ya es sabido, *ContactMap* requiere de varios componentes diferentes para poder funcionar correctamente. En concreto, se necesitan los siguientes elementos:

- El sistema gestor de bases de datos MySQL, con la base de datos *ContactMapBD* funcionando en él.
- El servidor web Apache Tomcat, donde se ha de publicar el servlet *ContactMapServlet*.
- La aplicación *ContactMap*, ejecutándose bien en un dispositivo móvil o en un emulador, a través del entorno de desarrollo Eclipse.

Todos los instaladores en su versión correspondiente, así como los ficheros que contienen scripts o fuentes, están incluidos en el CD adjunto a este proyecto.

## *7.1 Instalar MySQL*

Para realizar la instalación del sistema gestor de la base de datos MySQL se utilizará el asistente de instalación de MySQL. La versión utilizada aquí es la 5.1.30. Los pasos a seguir son los enumerados a continuación:

1. Descomprimir y ejecutar el instalador de nombre "mysql-5.1.30-win32.zip", presente en el CD adjunto.
2. Pulsar *Next* y escoger una instalación típica (opción *Typical*)
3. Pulsar *Install*.
4. Una vez terminada la instalación, pulsar en la opción *Configure MySQL Server now*.
5. Pulsar *Next* y elegir la opción *Detailed Configuration*. Pulsar de nuevo *Next*.
6. Escoger la opción *Server Machine* y pulsar *Next*.
7. Seleccionar la opción *Transactional Database Only*. Pulsar *Next*.
8. Pulsar de nuevo Next. En la siguiente pantalla se pregunta acerca del número de transacciones simultáneas que tendrá la base de datos. Elegir la más adecuada y pulsar *Next*.
9. En la siguiente pantalla, asegurarse de que las opciones *Enable TCP/IP Networking* y *Enable Strict Mode* estén seleccionadas. Además, donde pone *Port Number* debe estar introducido el valor 3306. Pulsar *Next*.
10. Seleccionar la opción *Standard Character Set*. Pulsar *Next*.
11. Marcar las opciones *Install As Windows Service*, *Launch the MySQL Server automatically* e *Include Bin Directory in Windows PATH*. Pulsar *Next*.

12. En la siguiente pantalla, señalar únicamente la opción *Modif. Security Settings*. Introducir la contraseña de administrador y pulsar el botón *Next*.[1]
13. Pulsar el botón *Excute*. Esto deberá finalizar exitosamente la configuración de MySQL (ver Figura 35). Si este paso produce algún error, es muy posible que sea debido al bloqueo del puerto asignado para MySQL, debiendo en tal caso revisar los firewall instalados.

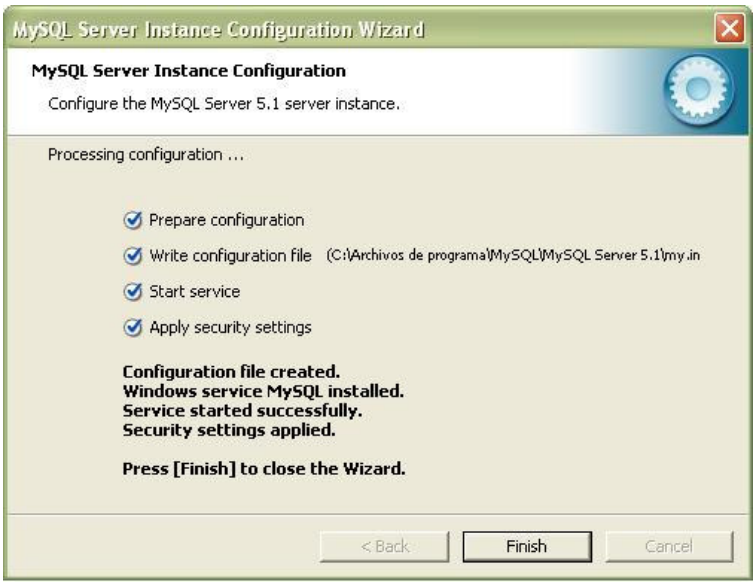


**Figura 35. Configuración exitosa de MySQL**

## 7.2 Cargar la base de datos ContactMapBD

Una vez instalado MySQL es necesario cargar en él la base de datos *ContactMapBD*. Esta consta de una sola tabla donde se almacenarán los datos que los diferentes usuarios de *ContactMap* pueden consultar y actualizar. Para poder cargar la base de datos, es necesario disponer de una cuenta de administrador en MySQL (la especificada durante su instalación):

1. Localizar el fichero "ContactMapBD.sql" en el CD adjunto.
2. Copiar dicho fichero en la carpeta de MySQL "\MySQL\bin".
3. Lanzar el prompt de MySQL y escribir la siguiente orden:

```
$> \. ContactMapBD.sql
```

[1] El servlet ContactMapServlet utiliza por defecto el usuario "root" con contraseña "rootroot". En caso de introducir otros datos para el administrador de MySQL, será necesario cambiar el fichero fuente del servlet "ContactMapServlet.java", presente también en el CD adjunto.

Llegados a este punto, la base de datos ya está creada y lista para funcionar con *ContactMap*. No es necesario incluir ningún dato inicial, aunque se pueden utilizar datos de prueba para comprobar el correcto funcionamiento de la aplicación. Para insertar datos de prueba, síganse los siguientes pasos:

4. Localizar el fichero "datosPrueba_ContactMapBD.sql" en el CD adjunto.
5. Copiar dicho fichero en la carpeta de MySQL "\MySQL\bin".
6. Lanzar el prompt de MySQL y escribir la siguiente orden:

```
$> \. datosPrueba_ContactMapBD.sql
```

Esta instrucción inserta 5 usuarios de prueba, cada uno con su número de teléfono identificativo, coordenadas y fechas, tal y como sugiere la siguiente tabla:

| Número | Latitud | Longitud | Fecha |
|---|---|---|---|
| 600100101 | 40.425060 | -3.681392 | 2009-01-17 18:30:00 |
| 600100102 | 40.438553 | -3.701897 | 2009-01-17 18:38:50 |
| 600100103 | 40.331882 | -3.766787 | 2009-01-17 15:12:34 |
| 600100104 | 40.333298 | -3.765521 | 2009-01-17 17:41:03 |
| 600100105 | 40.423985 | -3.674885 | 2009-01-17 16:01:14 |

**Tabla 5. Datos de prueba para la base de datos**

## 7.3 Instalar Apache Tomcat

El servidor web Apache Tomcat se instala a través del fichero instalador disponible en el CD. Apache Tomcat permite publicar el servlet que utiliza *ContactMap* para acceder a la base de datos instalada en el anterior apartado. Para instalar Apache Tomcat en su version 6.0.18 se deben seguir los siguientes puntos:

1. Localizar y ejecutar el fichero de nombre "apache-tomcat-6.0.18.exe" en el CD adjunto.
2. Pulsar *Next*, leer y aceptar las condiciones de uso mediante *I agree*.
3. Seleccionar instalación normal y pulsar *Next*.
4. Especificar la carpeta de instalación y pulsar *Next*.
5. Indicar el puerto de escucha y la contraseña de administrador. Pulsar *Next*.
6. Señalar la carpeta donde esté instalado el SDK de Java y pulsar *Install*.
7. La instalación debería finalizar exitosamente.


**Figura 36. Instalación exitosa de Apache Tomcat**

## 7.4 Publicar el servlet ContactMapServlet

El servlet *ContactMapServlet* hace de intermediario entre la aplicación *ContactMap* y la base de datos, permitiendo a la primera actualizar y consultar los datos de los diferentes usuarios. El servlet atiende peticiones HTTP llegadas al servidor, respondiendo en el mismo protocolo. Para colgar el servlet en el servidor Apache Tomcat deben realizarse los siguientes pasos:

1. Localizar en el CD el fichero de nombre "ContactMapServlet.war".
2. Copiar y pegar dicho fichero en la carpeta "\Tomcat 6.0\webapps".
3. Detener y reiniciar el servidor Apache Tomcat. Esto creará automáticamente la carpeta "\Tomcat 6.0\webapps\ ContactMapServlet"
4. Localizar en el CD el fichero de nombre "web.xml".
5. Copiar y pegar este fichero en la carpeta "\Tomcat 6.0\webapps\ContactMapServlet\WEB-INF"

Así pues, el servlet ya estaría disponible a través de Apache Tomcat, de forma que la aplicación *ContactMap* pueda consultar y actualizar en la base de datos. Puede que se desee además disponer de alguna herramienta que permita enviar peticiones en XML directamente al servlet, sin tener que utilizar *ContactMap* para ello:

1. Localizar en el CD el fichero de nombre "index.html".
2. Copiar y pegar el archivo en la carpeta "\Tomcat 6.0\webapps\ ContactMapServlet".

3. Para acceder, escribir en la barra de dirección del navegador "http://localhost:8080/ContactMapServlet/"[2].

Esta página web muestra un simple formulario HTML donde se pueden introducir directamente las peticiones XML al servlet, en el formato válido que este acepta (véase el capítulo correspondiente). Al enviar esta petición, la respuesta XML correspondiente a la misma se muestra de forma inmediata.

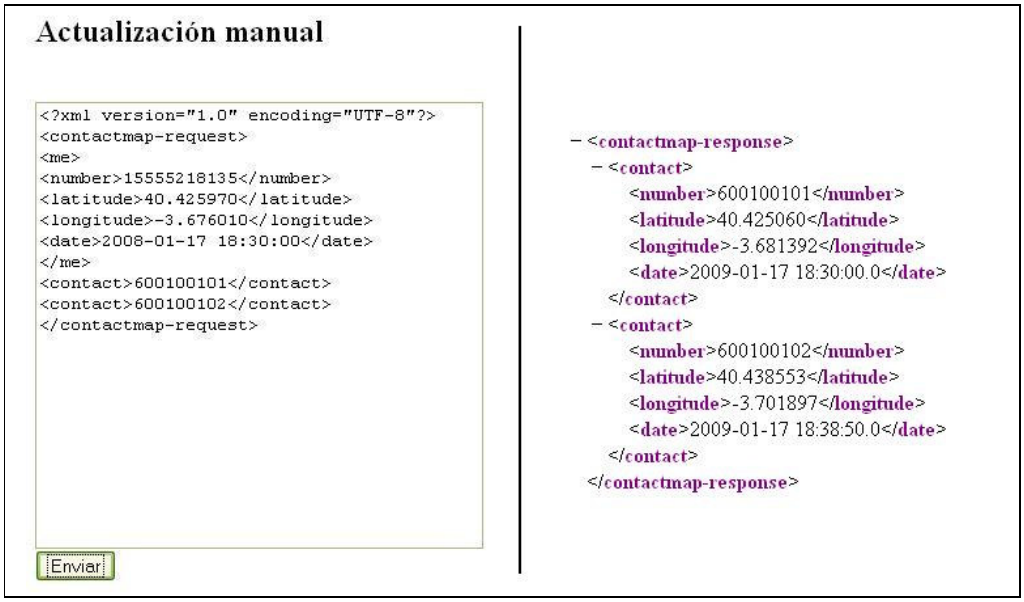


**Figura 37. Petición y su respuesta XML, a través del formulario HTML**

## 7.5 Instalar la aplicación ContactMap

Una vez que se ha instalado la base de datos y el servidor web con el servlet que posibilita el acceso a ella, ya se puede utilizar la aplicación *ContactMap* plenamente. El último paso, por tanto, es ejecutar dicha aplicación bien en un dispositivo móvil o bien en un emulador de Android.

### 7.5.1 Instalar en un dispositivo móvil

En el momento de la realización de este proyecto, únicamente existe a la venta un dispositivo móvil con Android incorporado, el modelo G1 de HTC. Además, su compra es posible en EEUU o Reino Unido, pero no en España (exceptuando el terminal de desarrollo ofrecido por Google [47]).

[2] Si se ha establecido en Apache Tomcat un puerto distinto al 8080, cambiar dicho valor.

Sobra decir que no se ha contado, por tanto, con ningún dispositivo físico para desarrollar o probar *ContactMap*, únicamente el emulador con el que cuenta el SDK de Android. Aún así, sí es posible dar algunos consejos acerca de la instalación de *ContactMap* en futuros modelos con Android:

- El único fichero necesario para la instalación de una aplicación Android es el .apk, resultante de la compilación de dicha aplicación. Dentro del proyecto para Eclipse llamado *ContactMap*, presente en el CD adjunto, se podrá encontrar el fichero de nombre "ContactMap.apk".

- Tal y como se explica en el capítulo correspondiente, *ContactMap* utiliza los servicios de Google Maps, para los cuales es necesario obtener una clave o *API Key* facilitada por Google. Hasta ahora, al utilizar solamente el emulador, la clave usada ha sido la de prueba, pero para poder instalarlo en un dispositivo móvil de verdad es necesario obtener una clave propia. Una vez obtenida una clave propia, el proyecto debe recompilarse especificándola tanto en las propiedades del proyecto como en el recurso externo "api_key", presente en la carpeta "\res\values\strings.xml" (consultar apartado 4.2.1 para más información).

- En el modelo G1 de HTC, único disponible con Android en el momento de redacción de estas líneas, es necesario indicar explícitamente en el dispositivo que se permite la instalación de aplicaciones fuera del servicio Android Market de Google. Además, se debe tener instalada también en el dispositivo la aplicación ADB Installer, que debería permitir con la sencilla instrucción "adb install ContactMap.apk" la instalación en el dispositivo de *ContactMap*.

### 7.5.2 Ejecutar desde el emulador de Android

Todo el presente proyecto se ha realizado utilizando el emulador incluido en el SDK de Android para desarrollar y probar la aplicación *ContactMap*. Dentro del CD adjunto puede encontrarse un proyecto para Eclipse con el nombre *ContactMap*. Para ejecutar la aplicación desde el emulador, solamente se necesita disponer de un entorno de desarrollo Eclipse con el *plug-in* de Android instalado, e importar dicho proyecto.

Para conocer cómo instalar Eclipse y el *plug-in* de Android, así como conocer qué hacer para ejecutar la aplicación, consúltese el tercer capítulo.

Al igual que con la base de datos, en el CD adjunto se dispone también de unos contactos de prueba para utilizar en el dispositivo móvil, en este caso a través del emulador. Dichos contactos se corresponden con los mismos introducidos como prueba en la base de datos. Estos permiten poder probar la aplicación *ContactMap* desde el primer momento. Los contactos son los siguientes:

| Nombre | Teléfono | Correo electrónico |
|---|---|---|
| Carlos | 600100101 | carlos@gmail.com |
| David | 600100102 | david@gmail.com |
| Elena | 600100103 | elena@gmail.com |
| Juan | 600100104 | juan@gmail.com |
| Susana | 600100105 | susana@gmail.com |

**Tabla 6. Datos de prueba para los contactos del dispositivo móvil**

Para poder disponer de estos datos, debe localizarse en el CD el archivo de nombre "userdata-qemu.img". Este archivo debe situar en el lugar configurado en las propiedades del proyecto para guardar y obtener datos que deberían ser almacenados en el dispositivo físico; por defecto, suele ser el directorio raíz.

# 8 ANEXO B: CONTENIDO DEL CD

Adjunto a esta memoria se encuentra un CD con diferentes contenidos relacionados con la instalación de la aplicación *ContactMap*, su documentación y otros elementos. Seguidamente, se exponen las carpetas y archivos guardados en dicho CD:

- `apache-tomcat-6.0.18.exe`: instalador de Apache Tomcat en su versión 6.0.18, utilizada en este proyecto.
- `ContactMap`: proyecto completo para Eclipse que contiene la aplicación *ContactMap*. Incluye los ficheros fuentes, el fichero compilado ".apk", los recursos externos y el manifiesto.
- `ContactMapBD.sql`: script para crear la base de datos *ContactMapBD*.
- `ContactMapServlet`: proyecto completo para Eclipse que contiene el servlet *ContactMapServlet*.
- `ContactMapServlet.war`: fichero de despliegue para colgar automáticamente el servlet *ContactMapServlet* en el servidor Apache Tomcat.
- `datosPrueba_ContactMapBD.sql`: script en SQL para insertar algunos usuarios de prueba en la base de datos.
- `index.html`: documento web para interactuar manualmente con el servlet *ContactMapServlet*.
- `mysql-5.1.30-win32.zip`: fichero comprimido que contiene el instalador de MySQL en su versión 5.1.30, la utilizada en este proyecto.
- `userdata-qemu.img`: imagen de datos para el emulador de Android, que contiene algunos contactos de prueba.
- `web.xml`: fichero XML que describe el despliegue del servlet ContatMapServlet.

# 9 ANEXO C: REFERENCIAS

[1] Android, el sistema operative de Google. Último acceso, enero de 2009. http://code.google.com/intl/es-ES/android/index.html

[2] Página oficial del HTC G1, primer dispositivo comercial con Android. http://www.htc.com/www/product/g1/overview.html

[3] "Motorola se cambia a Android", entrada publicada en el blog Gizmóvil sobre tecnologías y dispositivos móviles el 19 de diciembre de 2008. Última visita en diciembre de 2008. http://gizmovil.com/2008/12/motorola-se-cambia-a-android/

[4] "Programming for the Series 60 platform and Symbian", Digia Inc. Editorial Wiley, 2003.

[5] Canalys, web de análisis de mercados tecnológicos. Informe sobre el ejercicio 2007. Último acceso en octubre 2008. http://www.canalys.com/pr/2008/r2008021.htm

[6] "Symbian OS Communications Programming", Michael J. Jipping. Editorial Wiley, 2002.

[7] "Diseño y desarrollo de una herramienta de control y configuración de opciones de seguridad en Symbian", Daniel Marcos Martín. Proyecto Fin de Carrera, 2007.

[8] Symbian, teléfonos con sistema operativo Symbian. Último acceso en octubre 2008 http://www.symbian.com/phones/index.html

[9] "How to Do Everything with Windows Mobile", Frank McPherson. Editorial McGraw-Hill, 2006.

[10] "Diseño y desarrollo de una librería para la abstracción del componente RIL de Windows Mobile y aplicaciones de uso", Vanessa Martínez Oliva. Proyecto Fin de Carrera, 2008.

[11] MSDN, MS Windows CE Operating System Architecture. Último acceso en octubre 2008. http://www.microsoft.com/technet/archive/wce/plan/modular.mspx?mfr=true

[12] Gizmodo, blog sobre dispositivos electrónicos. Entrada publicada en enero de 2007. Último acceso en octubre 2008. http://gizmodo.com/gadgets/portable-media/windows-mobile-editions-get-less-confusing-names-professional-standard-and-classic-232300.php

[13] "Mobile Phone Programming and its Application to Wireless Networking", Frank H.P. Fitzek y Frank Reichert. Editorial Springer, 2007.

[14] "Java a Tope", Sergio Gálvez Rojas y Lucas Ortega Días. Editado por la Universidad de Málaga, 2003.

[15] Java World, artículo "The fragmentario effect". Último acceso en octubre 2008. http://www.javaworld.com/javaworld/jw-05-2004/jw-0524-fragment.html?page=1

[16] Noticias de Google en español. Artículo "Google cumple 10 años". Último acceso en octubre 2008.
http://google.dirson.com/post/4090-google-cumple-10/

[17] Público, edición digital. Noticia publicada el 11 de junio de 2008. Último acceso en octubre 2008.
http://www.publico.es/culturas/125240/google/obtiene/premio/principe/asturias/comunicacion/humanidades

[18] Servicios de Google Inc. Último acceso en octubre 2008.
http://www.google.es/options/

[19] El Mundo, edición digital. Noticia de la sección El Navegante, publicada el 27 de enero de 2006. Último acceso en octubre 2008.
http://www.elmundo.es/navegante/2006/01/26/empresas/1138270001.html

[20] El Mundo, edición digital. Noticia de la sección El Navegante, publicada el 6 de enero de 2004. Último acceso en octubre 2008.
http://www.elmundo.es/navegante/2004/04/06/empresas/1081242789.html

[21] Expansión, edición digital. Noticia publicada el 20 de octubre de 2008. Último acceso en octubre 2008.
http://www.expansion.com/2008/10/19/juridico/1224447142.html

[22] BBC News, noticia publicada el 4 de Julio de 2008. Último acceso en octubre 2008.
http://news.bbc.co.uk/hi/spanish/business/newsid_7489000/7489049.stm

[23] Condiciones de la Licencia Apache versión 2.0. Último acceso en septiembre de 2008
http://www.apache.org/licenses/LICENSE-2.0.html

[24] Página oficial del proyecto Android. Último acceso en septiembre de 2008
http://code.google.com/android/index.html

[25] Página oficial de Open Handset Alliance. Último acceso en septiembre de 2008
http://www.openhandsetalliance.com/index.html

[26] El Mundo, edición digital. Noticia de la sección El Navegante, publicada el 5 de noviembre de 2007. Último acceso en septiembre de 2008
http://www.elmundo.es/navegante/2007/11/05/tecnologia/1194284462.html

[27] Descripción de la arquitectura de Android, por Google. Último acceso en septiembre de 2008
http://es.youtube.com/watch?v=QBGfUs9mQYY

[28] Dalvik VM Internals, especificaciones de la máquina virtual. Último acceso en septiembre de 2008
http://sites.google.com/site/io/dalvik-vm-internals

[29] Barrapunto, artículo publicado el 5 de diciembre de 2007: "¿Por qué Google desarrolla Dalvik en vez de usar Java Micro Edition?". Último acceso en septiembre de 2008
http://softlibre.barrapunto.com/article.pl?sid=07/12/05/1357216

[30] Android Comunity, entrevista a Jason Chen, desarrollador de Android. Último acceso en septiembre de 2008
http://androidcommunity.com/jason-chen-answers-questions-about-android-20080603/

[31] Descripción del ciclo de vida de una aplicación Android, por Google. Último acceso en septiembre de 2008
http://es.youtube.com/watch?v=fL6gSd4ugSI

[32] SDK de Android. Web del proyecto Android. Último acceso en septiembre de 2008
http://code.google.com/android/download.html

[33] Descargas de distintos paquetes de Eclipse. Web oficial de la plataforma Eclipse. Último acceso en septiembre de 2008
http://www.eclipse.org/downloads/

[34] Guía de instalación del SDK de Android. Web del proyecto Android. Último acceso en septiembre de 2008
http://code.google.com/android/intro/installing.html

[35] "UML Distilled. A brief guide to the standard object modelling languague", Martin Fowler. Tercera edición, editorial Addison-Wesley.

[36] Finalistas del concurso "Android Developer Challenge". Último acceso en diciembre de 2008.
http://code.google.com/intl/es-ES/android/adc_gallery/

[37] Android Maps API Key Signup. Último acceso en dicimbre de 2008.
http://code.google.com/intl/es-ES/android/maps-api-signup.html

[38] MyGeoPosition, servicio de obtención de coordenadas. Última visita en diciembre de 2008.
http://www.mygeoposition.com

[39] Android Developers Blog, blog de los desarrolladores de Android. Último acceso en enero de 2009.
http://android-developers.blogspot.com/2008/08/some-information-on-apis-removed-in.html

[40] Infojobs Trends, salarios profesionales en España. Último acceso en enero de 2009.
http://salarios.infojobs.net

[41] PC Componentes, web de venta de hardware. Último acceso en enero de 2009.
http://www.pccomponentes.com

[42] Ebay, web de comercio electrónico. Último acceso en enero de 2009.
http://www.ebay.es

[43] Android-Spa, principal comunidad de desarrolladores de Android en español: noticas, manuales, foros, etc. Último acceso en enero de 2009
http://www.android-spa.com/

[44] anddev.org, comunidad de desarrolladores de Android (en inglés). Último acceso en enero de 2009. Último acceso en diciembre de 2008.
http://www.anddev.org/

[45] desarrolladores-android, comunidad en Google Groups de desarrolladores sobre Android en español.Último acceso diciembre de 2008.
http://groups.google.com/group/desarrolladores-android

[46] Android Developers, comunidad en Google Groups de desarrolladores para Androd (en inglés). Último acceso en diciembre de 2008.
http://groups.google.com/group/android-developers

[47] Devices for developers, iniciativa de Google para facilitar terminales a los desarrolladores. Última visita en enero de 2009.
http://code.google.com/intl/es-ES/android/dev-devices.html

# 10 ANEXO D: TÉRMINOS

- Acelerómetro: instrumento destinado a medir aceleraciones. Frecuentemente se utiliza para conocer la velocidad y desplazamiento de un cuerpo en uno o varios ejes dimensionales.

- Add-on: ver *plug-in*.

- AIDL: siglas de *Android Interface Definition Language*, en español Lenguaje para la Definición de Interfaces en Android. Constituye un lenguaje de sintaxis muy básica que permite describir interfaces que pueden ser utilizadas de forma remota.

- API: siglas de Application Programming Interface, en español Interfaz de Programación de Aplicaciones. Consiste en un conjunto de llamadas que ofrecen acceso a funciones y procedimientos, representando una capa de abstracción para el desarrollador.

- Background: representa un proceso que se ejecuta con pocos recursos, que no requiere interacción directa con el usuario y que existe sin el conocimiento de este.

- Biblioteca: agrupación de código y datos que proporcionan servicios a programas independientes, pasando a formar parte de éstos. Permiten la distribución de funcionalidades y la construcción modular. También conocido como librería, por su similitud con el inglés *library*.

- Bluetooth: protocolo que permite la transmisión de datos entre dispositivos, más o menos próximos y alineados, mediante un enlace de radiofrecuencia. Está especialmente diseñado para dispositivos de bajo consumo y coste.

- Bytecode: código intermedio, más abstracto que el código máquina, y que necesita de un mediador o máquina virtual para poder ser transformado y ejecutado en un hardware local.

- Callback: se denomina así a la relación que existe entre dos procesos cuando el origen de la comunicación es a su vez llamado o invocado por el proceso destino.

- Checkbox: elemento de interfaz de usuario que permite hacer una selección múltiple en un conjunto de opciones.

- Códec: abreviatura de *Compresor-Decompresor*, en español Compresor-Descompresor. Representa un algoritmo software o componente hardware que permite transformar un flujo de datos en una señal interpretable.

- Controlador: programa que permite al sistema operativo interactuar con un periféricos, abstrayéndolo y propocionando una interfaz para usarlo. También conocido como *driver*.

- CORBA: siglas de *Common Object Request Broker Architecture*, en español Arquitectura Común de Intermediarios en Peticiones a Objetos. Define un estándar mediante el cual elementos distribuidos programados en distintos lenguajes pueden intercambiar datos y objetos.

- CPU: siglas de *Central Processing Unit*, en español Unidad Central de Procesamiento. Elemento de hardware que controla el funcionamiento de un computador y lleva a cabo sus funciones de procesamiento de instrucciones y datos.

- CVM: siglas de *Compact Virtual Machine*. Representa, junto a KVM, una de las máquinas virtuales de Java disponibles en Java ME.

- Dalvik: nombre de la máquina virtual utilizada por el sistema operativo Android. Dalvik esta específicamente adaptada a las características de rendimiento de un dispositivo móvil y trabaja con ficheros de extensión ".dex", obtenidos desde el *bytecode* de Java.

- Datagrama: fragmento de información enviado por una red bajo un determinado protocolo de comunicación, generalmente formado por una cabecera y datos.

- Dispositivo móvil: aparato electrónico que es de reducido tamaño, cuenta con cierta capacidad tanto para la computación como para el almacenamiento de datos y cuenta con elementos de E/S básicos, por ejemplo pantalla y/o teclado.

- Driver: ver controlador.

- DTD: siglas de *Document Type Definition*, en español Definición de Tipo de Documento. Permite definir el formato que ha de tener un determinado documento XML.

- E/S: abreviatura de Entrada/Salida. Un elemento de E/S es aquél que permite la comunicación entre un sistema de procesamiento de datos y una entidad externa a él (un usuario humano u otro sistema de procesamiento). Entrada se considera toda aquella información que es recibida por un sistema, mientras que salida es aquella información que es enviada por el mismo.

- Ebook: dispositivo electrónico que permite la visualización de documentos, como por ejemplo libros, en formato digital a través de una pantalla.

- EDGE: siglas en inglés de *Enhanced Data rates for GSM of Evolution*, en español Tasas de Datos Mejoradas para la evolución de GSM. Es una tecnología para teléfonía móvil que representa un puente entre la segunda y tercera generación de estos dispositivos.

- Fragmentación: fenómeno que describe una situación en la que una misma tecnología ha evolucionado de forma que se hace incompatible entre sí.

- Framework: término con el que se define un amplio conjunto de elementos que permite desarrollar y organizar software utilizando un determinado lenguaje, sistema o tecnología. Habitualmente incluye bibliotecas, programas de desarrollo o manuales.

- Googol: término con el que se designa al número 10 elevado a 100.

- GPS: siglas de *Global Positioning System*, en español Sistema de Posicionamiento Global. Es un sistema global de navegación que, mediante satélites, permite ubicar un objeto en la superficie terrestre con una precisión que va desde varios metros a centímetros.

- GSM: siglas de *Groupe Spécial Mobile*, más conocido como Sistema Global para las Comunicaciones Móviles, es el estándar más extendido para las comunicaciones con telefonía móvil. Permite llamadas, navegación por Internet o envío de SMS.

- GPRS: siglas de *General Packet Radio Service*, en español Servicio General de Paquetes vía Radio. es una extensión del estándar GSM que permite mejorar sus prestaciones originales, como el envío de datos, uso de correo electrónico, navegación web o el aumento de las tasas de transferencia de datos.

- GUI: siglas de *Graphical User Interface*, en español Interfaz Gráfica de Usuario. Representa la parte del software que, mediante un contexto o lenguaje principalmente visual y simbólico, permite al usuario utilizar una aplicación.

- Hilo: en sistemas operativos, un hilo constituye cada uno de los flujos de ejecución en el que puede ser divido un proceso. Todos los hilos de un proceso comparten espacio en memoria, archivos abiertos, variables globales, semáforos, etc. Permiten la ejecución concurrente de varias tareas. También llamado *thread*.

- HTTP: siglas de *HyperText Transfer Protocol*, en español Protocolo de Transferencia de Hipertexto. Constituye el protocolo utilizado para la transmisión de documentos a través de la Web entre un cliente y servidor.

- Interfaz: en computación, una interfaz se refiere a una abstracción que una determinada elemento ofrece de sí mismo al exterior, facilitando de esta forma su acceso y uso por otros elementos de hardware o software.

- JAD: siglas de *Java Aplication Descriptor*, en español Descriptor de Aplicación Java. Archivo que acompaña a una aplicación Java ME y que ofrece información general y de despliegue sobra la misma.

- JAR: acrónimo de *Java ARchive*, en español Archivo Java. Representa una agrupación de varios ficheros Java y se usa generalmente para la distribución conjunta de clases y metadatos.

- Java ME: Java Micro Edition, edición de Java especialmente dirigida a los dispositivo móviles.

- Java Micro Edition: ver Java ME.

- JSR: siglas de Java Specification Request, en español Solicitud de Especificación Java. Representa un documento formal que describe las especificaciones y tecnologías propuestas para ser añadidas oficialmente a la plataforma Java.

- JVM: siglas de *Java Virtual Machine*, en español Máquina Virtual de Java. Constituye un elemento software de la tecnología Java, encargado de transformar el código intermedio universal o *bytecode* en código máquina específico del hardware donde está instalado.

- Kernel: parte fundamental de un sistema operativo, responsable de facilitar acceso seguro al hardware, gestionar recursos y hacer llamadas al sistema. También conocido como núcleo.

- KVM: siglas de *Kilobyte Virtual Machine*. Representa, junto a CVM, una de las máquinas virtuales de Java disponibles en Java ME.

- Laptop: ordenador portátil. También llamado *notebook*.

- Latitud: distancia angular entre el ecuador y un punto de la superficie del planeta. Se mide en grados entre 0 y 90.

- Librería: ver biblioteca.

- Listener: objeto que está a la espera de determinado evento.

- Longitud: distancia angular entre el meridiano y un punto de la superficie del planeta. Se mide en grados entre 0 y 360.

- Máquina virtual: representa un software que emula el comportamiento de una determinada arquitectura o que permite adaptar un código fuente a las características de la máquina nativa.

- Microkernel: tipo de kernel de un sistema operativo que provee un conjunto de primitivas o llamadas al sistema mínimas. También llamado micronúcleo.

- Micronúcleo: ver microkernel.

- Middleware: capa de abstracción software que posibilita el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas.

- MMS: siglas de *Multimedia Messaging System* o en español Sistema de Mensajería Multimedia. Es un estándar de mensajería que le permite a los teléfonos móviles enviar y recibir contenidos multimedia, incorporando sonido, video, fotos o cualquier otro contenido disponible en el futuro.

- MOAP: siglas de *Mobile Oriented Applications Platform*, una plataforma software basada en Symbian para los teléfonos del fabricante japonés FOMA.

- MP3: formato de compresión de audio digital cuyo nombre completo es MPEG-1 Audio Layer 3. También utilizado, por extensión, para nombrar al dispositivo móvil reproductor de audio digital en el que se almacena, organiza o reproduce archivos de audio digital.

- Núcleo: ver kernel.

- OHA: siglas de *Open Handset Alliance*, un conglomerado de empresas de sectores tecnológicos lideradas por Google que promueve la innovación y desarrollo de dispositivos móviles y sus aplicaciones. Su primera contribución es el sistema operativo Android.

- PDA: siglas de *Personal Digital Assistant*, en español Asistente Digital Personal. Dispositivo móvil utilizado como organizador personal, que cuenta generalmente con pantalla táctil, agenda, calendario, conectividad Wi-Fi, y aplicaciones ofimáticas, entre otros.

- Plug-in: componente de software que se relaciona y ejecuta con otro para aportarle una función nueva y generalmente muy especifica.

- Proceso: un proceso es un programa en ejecución, y representa la unidad de procesamiento básica gestionada por el sistema operativo.

- Radiobutton: elemento de interfaz de usuario que permite seleccionar un único elemento dentro un conjunto definido de valores.

- RAM: siglas de *Random Access Memory*, o en español Memoria de Acceso Aleatorio. Componente de memoria volátil, con palabras de acceso individual mediante una dirección de memoria específica y con rápida ejecución de operaciones de lectura y escritura.

- RMI: siglas de Remote Method Invocation, en español Invocación de Métodos Remotos, es una tecnología de Java que permite comunicar objetos distribuidos escritos en este lenguaje.

- RMS: siglas de *Record Management System*, en español Sistema de Gestión de Registros. Representa un mecanismo de almacenamiento permanente en dispositivos con Java ME.

- SAX: siglas de *Simple API for XML*, en español API Simple para XML, representa una conocida API para Java que facilita el procesamiento de documentos XML.

- SDK: siglas de Software Development Kit, en español Kit de Desarrollo de Software. Constituye un conjunto de herramientas que permiten a un desarrollador crear aplicaciones para una determinada plataforma o lenguaje.

- Servlet: elemento de la tecnología Java, que extiende la funcionalidad de un servidor Web, aceptando y procesando peticiones.

- Sistema operativo: programa cuya finalidad principal es simplificar el manejo y explotación de un elemento con capacidad computacional, gestionando sus recursos, ofreciendo servicios a las demás aplicaciones y ejecutando mandatos del usuario.

- *Smartphone*: dispositivo móvil que representa una evolución de los teléfonos móviles, con la inclusión de pantalla táctil, teclado, conexión Wi-Fi, aplicaciones de usuario como navegador web o cliente de correo, entre otros.

- SMS: siglas de *Short Message Service*, en español Servicio de Mensajes Cortos, es un estándar de la telefonía móvil que permite enviar mensaje de texto con un número de caracteres limitado.

- Socket: abstracción software, identificada por una dirección IP, un protocolo y un puerto, que permite la comunicación de dos programas, generalmente situados en computadores distintos.

- Teclado QWERTY: teclado cuya distribución de letras es la más común hoy día en ordenadores y otros elementos de computación. Toma su nombre de sus 5 primeras letras alfabéticas: Q, W, E, R, T, e Y.

- Thread: ver hilo.

- UIQ: siglas de *User Interface Quartz*, una plataforma software basada en Symbian usada en algunos teléfonos de los fabricantes Sony Ericcson y Motorola.

- UMTS: siglas de *Universal Mobile Telecommunications System*, en español Sistema Universal de Telecomunicaciones Móviles. Constituye en estándar de comunicación para dispositivos de tercera generación o 3G, que ofrece capacidades multimedia y conexiones de alta velocidad en Internet.

- WAP: siglas de *Wireless Application Protocol*, en español o Protocolo de Aplicaciones Inalámbricas. Es un estándar para aplicaciones que utilizan las

comunicaciones inalámbricas, como el acceso a servicios de Internet desde un teléfono móvil.

- Widget: componente gráfico utilizado en interfaces de usuario, con el cual el usuario puede interactuar, como por ejemplo cajas de texto, botones, ventanas, etc.

- Wi-Fi: acrónimo de *Wíreless Fidelity*, estándar de envío de datos que utiliza ondas de radio en lugar de cables.

- XML: siglas de *Extensible Markup Language*, en español Lenguaje de Marcado Extensible. Representa un lenguaje estándar que, mediante el uso de etiquetas y atributos, permite expresar e intercambiar fácilmente estructuras de datos.

- XMPP: siglas de *Extensible Messaging and Presence Protocol*, en español Protocolo Extensible de Mensajería y Presencia. Es un protocolo basado en XML que se utiliza en servicios de mensajería instantánea.