ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Diseño e implementación de un juego Android para soporte a la enseñanza de frameworks OO y patrones de diseño.

AUTOR: RAÚL ILLÁN GARCIA

DIRECTOR: JUAN ANGEL PASTOR FRANCO

Noviembre **/** 2012

INDICE

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Planteamiento inicial

Se parte de un prototipo de juego implementado utilizando las librerías *Javax.swing*.

El juego consiste en naves que compiten entre sí por alcanzar un determinado objetivo en un entorno virtual. Cada nave dispone de una pantalla de radar y de un mapa parcial (inicialmente vacío) del entorno.

El juego puede desarrollarse en dos modos: interactivo o automático. En el modo interactivo el jugador tiene el control. En el modo automático la nave es autónoma y se guía por una serie de algoritmos de navegación, control y reacción (p.e.: evitación de obstáculos).

El objetivo general es refactorizar el diseño del juego y trasladarlo a la plataforma Android para ser utilizado con fines docentes para explicar el diseño de *frameworks* en general y de Android en particular.

1.2 Objetivos

Adaptación del juego original a Android de forma que pueda servir para preparar docencia.

Extensión del juego original con una pantalla de radar y posibilidad de ejecución simultánea de varios jugadores desde distintas plataformas móviles Android.

Documentación del proyecto de forma que pueda usarse con fines docentes.

Se plantea una aplicación con diferentes formas de funcionamiento:

- Modo automático
- Modo manual
- Modo de juego en red.

Cada uno de los modos de funcionamiento servirá además para explicar alguna característica o conjunto de características de la aplicación. De esta manera:

- El modo de funcionamiento automático se ha hecho para explicar como un juego actualiza los datos y redibuja la pantalla, sin la intervención del usuario, y como funciona Android con respecto a las vistas XML, captura de eventos en la interface visual y demás características sobre la programación en Android.

- El modo de funcionamiento manual se utiliza para explicar la utilización de sensores, como el acelerómetro.

- El modo de funcionamiento en red se utiliza para explicar la utilización de socket, en concreto socket UDP, en Android. Esta versión tiene un servidor que se ejecuta en un PC y un cliente que se ejecuta en el dispositivo Android.

Se explicaran con más profundidad más adelante.

1.3 **Requisitos**

El cliente deberá disponer de la siguiente funcionalidad:

- Para la versión automática, el dispositivo debe disponer una versión de SO android compatible (Android 4.0 o superior).

- Para la versión manual, el dispositivo deberá disponer de una versión de SO android compatible (Android 4.0 o superior) y de sensor de aceleración para captura la posición del dispositivo y así ofrecer movimiento a las naves virtuales.

Al ejecutar estas dos versiones se mostrará la siguiente interfaz grafica:

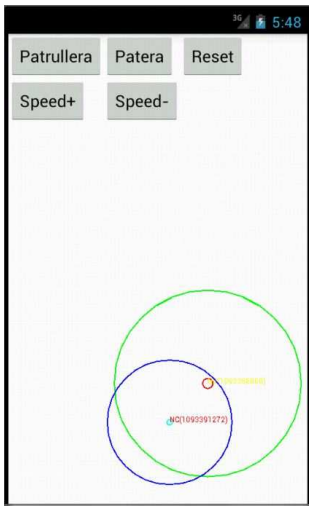

Figura 1 Interfaz grafica de modos automatico y manual

- Para la versión en red, el dispositivo deberá disponer de una versión de SO android compatible (Android 4.0 o superior), de sensor de aceleración, conexión wifi, y estar conectado a la misma red wifi que el servidor.

Al ejecutar esta versión aparecerá la siguiente interfaz:

Figura 2 Interfaz grafica de modo en red

El cliente deberá disponer de la siguiente funcionalidad:

- Deberá estar conectador a la misma red (por cable o wifi) que el cliente.
- La ip del servidor es fija, la cual corresponde con `192.168.1.14`

Al ejecutar el servidor en el PC aparecerá la siguiente ventana:

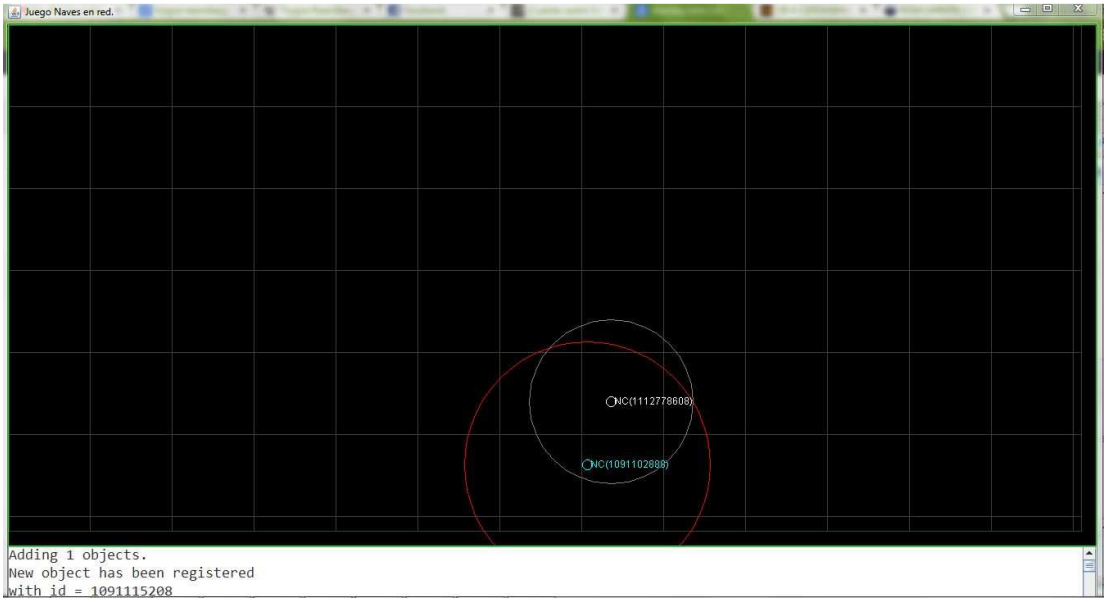

Figura 3 Interfaz grafica servidor modo red

2 Estado del arte

2.1 Introducción

El mundo de las aplicaciones móviles está viviendo hoy en día, sin duda, su momento más dulce.

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

Figura 4. Un portátil Apple Powerbook G4 de 2005 (izquierda) tiene la misma velocidad de procesador, cantidad de memoria RAM y espacio de almacenamiento que un teléfono móvil Samsung Galaxy S2 de 2011 (derecha).

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

aplicaciones para ir de compras, para navegación mediante GPS, para convertir el teléfono móvil en un punto de acceso WiFi, aplicaciones para leer la Biblia, para calcular los transbordos en el metro, para retoque fotográfico, aplicaciones para leer los periódicos extranjeros, para consultar los síntomas de una enfermedad... Aplicaciones todas ellas, que toman ventaja de las velocidades de transmisión de las redes actuales, de la posibilidad de conocer la localización del usuario, de la capacidad de proceso de los nuevos modelos y de la ubicuidad del acceso a internet y a toda la información que alberga, para ofrecer al usuario la que necesita en en ese momento.

2.2 **Definición y orígenes**

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

Figura 5. Dispositivo PDA, modelo Palm III de 1999, desarrollado por Palm Computing. Pantalla táctil LCD con 4 tonos de gris, 8 MB de memoria RAM. La imagen muestra el dispositivo acoplado a su base de conexión al PC.

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

2.3 **Telefonía móvil: J2ME y WAP**

Lo que no sucedió con las PDA, sucedió con los teléfonos móviles: en la segunda mitad de los 90, la introducción de las tecnologías GSM y EDGE con su sistema de mensajes cortos SMS, el protocolo WAP, el abaratamiento de costes y la liberalización del mercado de operadores se unieron para provocar un boom en la utilización de telefonía móvil, pasando sólo en España de 1 millón de usuarios en 1996 a casi 30 millones en 2001. Aunque los primeros terminales de 2ª generación (GSM) se limitaban a ofrecer la funcionalidad básica de comunicación por voz, envío de mensajes SMS y listín telefónico —con pantallas monocromas de dos o tres líneas de texto y, en muchos casos, ni siquiera letras minúsculas; aunque había algunas excepciones como el Nokia 9000 Communicator que, con sus aplicaciones de correo y navegación web, sólo estaba al alcance de los bolsillos más pudientes—, ya a finales de los 90 empezaron a aparecer los primeros móviles destinados al público general que incorporaban aplicaciones sencillas de tipo calendario, reloj y agenda, e incluso juegos. Sin embargo, en todos los casos eran aplicaciones propietarias incluidas por los propios fabricantes; los terminales de entonces no tenían ningún mecanismo sencillo que permitiera la instalación de aplicaciones de terceros.

Figura 6. De izquierda a derecha: en 1997, el modelo de Alcatel "One Touch Easy", con sus dos líneas de texto, no permitía margen para aplicaciones; en 1998, el modelo "5110" de Nokia es uno de los primeros en incorporar aplicaciones de entretenimiento

El éxito de los terminales que incorporaban juegos y aplicaciones a su funcionalidad predeterminada, provocó que los fabricantes reorientaran parte de sus líneas de producción a crear dispositivos orientados un poco más hacia el entretenimiento. Un cambio de mentalidad que abrió el mercado de la telefonía a un público joven que exigía constantemente más novedades, más tecnología, más variedad, más posibilidades, y que hizo avanzar el mercado a un ritmo vertiginoso. Con el cambio de siglo llegaron al público general nuevos terminales, más pequeños, con más capacidad y con pantallas de miles de colores, algunos de los cuales incorporaban dos tecnologías que, juntas, formaban el caldo de cultivo perfecto para el mercado de las aplicaciones: J2ME y WAP. Java 2 Mobile Edition (J2ME, hoy conocido como JME) surgió en 2001 como un subconjunto básico de las librerías y API de Java, diseñado para ser ejecutado en dispositivos embebidos como sistemas de control industrial, pequeños electrodomésticos y, por supuesto, terminales móviles y PDA. Su llegada al mundo de lo inalámbrico facilitó enormemente el desarrollo de aplicacionespara este tipo de dispositivos, puesto que ya no era necesario conocer lenguajes específicos de cada dispositivo o fabricante ni disponer obligatoriamente de un terminal en el que hacer las pruebas: los programadores podían crear sus aplicaciones cómodamente en su ordenado personal, con un emulador y usando un lenguaje fácil y potente como Java. J2ME abrió las puertas de los terminales móviles a los desarrolladores. WAP (Wireless Application Protocol), por otra parte, es un conjunto de protocolos que facilita la interoperabilidad de dispositivos móviles en redes inalámbricas como GSM y CDMA. Aunque comenzó a desarrollarse en 1997, fue a partir de 2001 cuando empezó a extenderse su uso, con la introducción de la funcionalidad "WAP Push". Esta nueva característica permitía a los operadores enviar a sus usuarios notificaciones con enlaces a direcciones WAP, de manera que el usuario sólo tenía que activarlas para acceder a contenidos en red. Gracias a esto, los terminales móviles podían

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

Figura 7. Fragmento de una página de revista, anunciando multitud de contenidos multimedia accesibles mediante WAP y mensajes premium.

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

2.3.1 Symbian

En el año 2000, la compañía Ericsson provocó un salto cualitativo en el mercado de los dispositivos móviles: presentó su modelo Ericsson R380 Smartphone, el primer teléfono móvil con pantalla táctil y el primero que integraba toda la funcionalidad de una auténtica PDA manteniendo al mismo tiempo el tamaño, peso y diseño de los terminales de entonces. Aunque la interfaz se parecía demasiado a la de las PDA de la época y a pesar

de que era un modelo "cerrado" que no permitía instalar aplicaciones de terceros, fue muy bien recibido por el público más técnico.

Figura 8. Teléfono móvil Ericsson modelo R380 (2000). Fue el primero en combinar un teléfono móvil con las funcionalidades de una PDA.

Este avance fue gracias a que el R380 usaba una versión de un sistema operativo, EPOC, que ya utilizaban algunas PDA de la época y que fue adaptado para su uso en terminales móviles. Dicha versión, conocida como Symbian, comenzó a desarrollarse en 1998 por una asociación conjunta de los fabricantes Nokia, Ericsson, Motorola y Psion, que tenían como objetivo aprovechar la incipiente convergencia entre teléfonos móviles y PDA para desarrollar un sistema operativo abierto que facilitara el diseño de los futuros dispositivos. El Ericsson R380 fue el primer modelo que salió al mercado con este sistema operativo y, ante el éxito cosechado y aprovechando el progreso de la tecnología utilizada en los terminales más recientes, Nokia y Ericsson enfocaron parte de sus esfuerzos a suplir las carencias de Symbian. Nokia, comprendiendo el potencial del nuevo sistema operativo, decidió utilizarlo en sus modelos "Communicator" que, a pesar de ser los más avanzados (y caros) de toda su gama —ya desde varios años antes incorporaban acceso a Internet, correo, un teclado completo y varias otras aplicaciones—, estaban hechos sobre una plataforma que empezaba a quedarse pequeña. Con su modelo Communicator 9210, Nokia introdujo a mediados del año 2001 en el mercado la versión 6 de Symbian OS, con multitarea real y acompañado de la nueva interfaz gráfica "Series 80" a todo color. El modelo fue muy bien recibido en algunos sectores de público que no estaban familiarizados con las PDA y para los cuales el Communicator suponía una novedosa solución "todo en uno" para sus negocios.

Figura 9. Teléfono móvil Nokia, modelo "Communicator" 9210 (2001), con teclado incorporado. Fue el primero en incorporar Symbian 6.0 y la interfaz "Series 80".

Sin embargo, el 9210 carecía de pantalla táctil y esto fue visto como un paso atrás por muchos clientes, que estaban acostumbrados a las pantallas táctiles de las PDA y que esperaban que un modelo con características de PDA como era el 9210 también tuviera una. Además, su tamaño de "ladrillo" lo dejaba fuera de la tendencia a la reducción de tamaño que estaba experimentando el mercado generalista de dispositivos. Nokia no estaba dispuesto a tirar la toalla y en el segundo cuarto de 2002 puso sobre la mesa su modelo 7650: el primer smartphone auténtico, con cámara incorporada y con un diseño agradable y tamaño similar al del resto de teléfonos móviles del mercado. Traía Symbian OS 6.1 y una interfaz gráfica, "Series 60", diseñada específicamente para terminales con pantalla y teclado "cuadrados", que era el resultado del trabajo de Nokia sobre la anterior "Series 80" para facilitar al máximo su uso. Y lo más importante de todo: permitía instalar aplicaciones de terceros.

Figura 10. Teléfono móvil Nokia 7650 (2002). Con Symbian OS 6.1 y 3,6 MB de espacio adicional, permitían instalar aplicaciones como por ejemplo una versión del navegador Opera.

Este modelo de Nokia fue un éxito instantáneo y el culpable, en gran medida, de que el mercado de las aplicaciones móviles se llenara de programas compatibles con dispositivos basados en S60. La posibilidad de instalar aplicaciones tanto nativas como basadas en J2ME, y de hacer cosas como comprobar el correo, hacer fotos o jugar a un juego, todo ello con una sola mano y utilizando un teléfono elegante que cabía cómodamente en el bolsillo, fue la mejor carta de presentación posible para la plataforma S60 de Nokia y de hecho, hasta 2010, S60 fue la interfaz más utilizada en todo el mercado de terminales móviles. Sin embargo, todavía hacían falta un par de componentes más para desencadenar la siguiente explosión en el mercado de las aplicaciones móviles, y uno de ellos era el espacio disponible. El Nokia 7650, con sus 3,6 MB de espacio, apenas dejaba espacio para aplicaciones muy elaboradas, y ni siquiera un fichero MP3 de tamaño medio podía ser almacenado en él. Ericsson, para entonces ya Sony Ericsson, se encargó de dar el siguiente paso en la evolución de Symbian, sacando al mercado su modelo P800.

Figura 11. Teléfono móvil Sony Ericsson P800 (finales de 2002), con teclado desmontable y pantalla táctil.

Este teléfono móvil fue el primero en traer Symbian 7.0, la siguiente versión del sistema operativo, así como una nueva interfaz gráfica desarrollada por Ericsson (antes de su fusión con Sony) y conocida como UIQ, que permitía instalar aplicaciones desarrolladas por terceros bajo C++ o bajo J2ME. El Sony Ericsson P800, además de su pantalla a todo color y su cámara VGA, llegaba acompañado de 16 MB de espacio pero podía, aprovechando la tecnología de almacenamiento mediante tarjetas SD desarrollada por Sony, ampliarse hasta unos increíbles (para entonces) 128 MB de espacio, lo que permitía al usuario utilizar su teléfono también como reproductor de música y hasta de vídeo. A partir de este momento, la evolución y expansión del sistema operativo Symbian sería, durante algunos años, imparable. Para los fabricantes, especialmente para Nokia y Sony Ericsson, era más fácil integrarlo en sus terminales al ser un sistema abierto; para los usuarios, era el sistema "de moda", el más compatible con todas las aplicaciones disponibles y el que más cosas podía hacer: varios modelos disponían de navegador web totalmente compatible con HTML, no solo WAP, y también aparecieron modelos con funcionalidades y diseño ideados para actividades específicas como escuchar música o jugar a videojuegos, conquistando así cuota de mercado en nichos tradicionalmente apartados del mundo de la telefonía. Y aunque los teléfonos más económicos siguieron utilizando sistemas cerrados específicos para cada modelo (casi siempre compatibles con J2ME); y a pesar de la entrada en el mercado de otros competidores que daban un gran valor añadido al servicio de telefonía móvil —como es el caso del fabricante Research In Motion con sus dispositivos BlackBerry, de gran éxito en los países norteamericanos, que permitían mensajería instantánea y servicios de correo electrónico encriptado gracias a su red privada de servidores—; Symbian llegó a copar el mercado mundial de teléfonos

móviles, con un 76% de cuota de mercado en 2006. Aún en 2008, más de la mitad de smartphones a la venta estaban basados en una versión u otra de Symbian.

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

Tabla 1 Cuota de mercado global de los principales sistemas operativos para terminales smartphone, de 2005 a 2008 [8]. Fuentes: Canalys, Gartner.

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

2.3.2 IPhone y Android

Hemos empezado este capítulo hablando de las PDA. Hasta 1992, nadie había oído nunca hablar de una PDA. Sí, existían dispositivos portátiles que ofrecían una ayuda para organizar tareas y tomar notas; pero no fue hasta ese año que Apple acuñó el término personal digital assistant (PDA) para referirse a su última creación: el Apple MessagePad, la primera PDA auténtica, con un nuevo sistema operativo (Newton), pantalla táctil, reconocimiento de escritura y un aspecto e interfaz que serían después imitados por todos los modelos de PDA del mercado. Hemos empezado este capítulo, por tanto, hablando de Apple; y es curioso, porque vamos a terminarlo hablando de Apple otra vez. Apple, que a principios de los 90 había revolucionado el mercado de dispositivos móviles con la introducción de sus modelos basados en el sistema operativo Newton, no cosechó sin embargo el éxito que esperaba con ellos. Una autonomía muy limitada por el uso de pilas AAA, un sistema de reconocimiento de escritura defectuoso que requería demasiado tiempo de "aprendizaje" y la falta de conectividad inicial con el ordenador del usuario (los cables y software necesario se comercializaban aparte) fueron las principales causas de que, durante la década de los 90, el mercado fuera dominado por otras compañías con modelos más depurados y Apple se centrara más en su línea de ordenadores personales

(llegando a cosechar enorme éxito a finales de siglo con su línea de ordenadores iMac y PowerBook y con su sistema operativo Mac OS).

Figura 12. A la izquierda, Apple MessagePad (1992), la primera en ser llamada PDA. A la derecha, Apple iPhone (2007), el detonador de la explosión del mercado de aplicaciones móviles.

Sin embargo, la mayor revolución de todas, la que en poco tiempo iba a convertir el mundo de las aplicaciones móviles en un negocio de miles de millones de dólares, llegaría de la mano de Apple con el cambio de milenio... aunque por una vía poco convencional: con música. Según declaraciones de Steve Jobs, CEO de Apple, por aquel entonces el comité ejecutivo de la compañía no creía que las PDA o los tablet PC (ordenadores portátiles con pantalla táctil que permitía su uso sin teclado, como si fueran una tabla) fueran buenas opciones para conseguir un gran volumen de negocio para Apple. Jobs creía que el futuro del mercado estaba en los teléfonos móviles; que éstos iban a hacerse dispositivos importantes para el acceso a la información de forma portátil, y que por tanto los teléfonos móviles debían tener una sincronización de software excelente. Por ello, en lugar de dedicarse a evolucionar su línea de PDA basada en el sistema Newton, los empleados de Apple pusieron todas sus energías en el reproductor de música iPod y la plataforma iTunes, un conjunto de software y tienda en línea que permitía a los usuarios del dispositivo la sincronización del mismo con su biblioteca de música así como la compra de nuevas canciones y discos de una manera fácil e intuitiva. El éxito de la plataforma iTunes fue demoledor. No sólo rompió todos los esquemas de lo que hasta entonces había sido el negocio de las discográficas, liberalizando el mercado y cambiando para siempre el paradigma de la venta de música y otras obras con propiedad intelectual. Fue además la demostración más clara de que cuantos menos impedimentos se le pongan al usuario para que compre algo, más fácil es que lo compre; y la primera piedra de la estructura que faltaba en el mercado de las aplicaciones móviles para convertirlas en el negocio que son hoy en día: una plataforma de distribución que permita a los programadores y empresas publicar y dar visibilidad a sus aplicaciones reduciendo

al máximo la inversión necesaria, y a los usuarios adquirir las aplicaciones de la manera más sencilla posible. Ya en 2008, Apple creó dentro de la plataforma iTunes la App Store, una tienda en línea que permitía a los usuarios comprar y descargar directamente en el móvil o a través del software iTunes aplicaciones para el producto estrella de la compañía, el teléfono móvil iPhone. Con una política de aceptación muy estricta, los desarrolladores que consiguen cumplir todos los requisitos exigidos por Apple pueden publicar en esta tienda sus aplicaciones y obtener el 70% de la recaudación de su venta. Con estas condiciones, muchos desarrolladores adaptaron su trabajo y sus aplicaciones a la plataforma iOS de Apple, basada en el lenguaje Objective-C y en entornos de desarrollo sólo disponibles para sistemas Mac Os, comercializados también por Apple. El mercado de las aplicaciones móviles disponía por fin de todos los elementos necesarios para su eclosión final, y así inició su despegue imparable, arropado por dispositivos con la tecnología necesaria para posibilitar aplicaciones interesantes y por plataformas de distribución de las mismas a un coste prácticamente cero. No obstante, la mejor parte del pastel, la que más beneficios generaba, estaba reservada a aquellos que abrazaran las tecnologías de Apple, y además era una parte forzosamente pequeña ya que no muchos usuarios podían permitirse adquirir dispositivos como el iPhone, con un coste de varios cientos de dólares. Por este motivo, en 2008 el monopolio de Symbian aún no peligraba.

Afortunadamente, ese mismo año 2008 llegó la alternativa: Android. Mientras que el negocio de Apple estaba limitado a un único modelo y a una tecnología y lenguaje propietarios, Google se presentaba con un sistema operativo abierto, basado en Linux y Java y que podía ser incluido en cualquier dispositivo móvil independientemente del fabricante. Mediante numerosos acuerdos comerciales, Google consiguió que varios de los más importantes fabricantes de teléfonos móviles, que hasta ese momento estaban vendiendo sus terminales con sistemas operativos propietarios o con distintas versiones de sistemas Symbian o Microsoft, adoptaran el nuevo sistema operativo Android: un sistema operativo moderno, respaldado por una gran empresa como era Google y que les permitiría competir con el rey del mercado, el iPhone, ahorrándose de paso los costes de desarrollo del sistema. Además, Android llegaba de la mano de una plataforma de distribución de aplicaciones, Google Play, similar a la de Apple pero unas políticas de aceptación mucho menos restrictivas. Con estas condiciones y con un entorno de desarrollo basado en el lenguaje Java y en herramientas disponibles gratuitamente tanto para sistemas Windows como para sistemas Linux/Unix, muchos desarrolladores y empresas con experiencia previa en Java se decantaron por Android como punto de entrada al mercado de las aplicaciones móviles. Desde entonces, el ascenso de Android ha sido imparable. Si en 2008 tenía menos de un 1% de mercado, en 2011 ha conseguido desbancar a Symbian como sistema operativo más utilizado ensmartphones, con un 53% del mercado.

2.4 **Escenario actual**

Hoy en día, el mercado de las aplicaciones móviles se reparte principalmente entre Apple con su sistema iOS y Google con su sistema Android. En el caso de iOS, con tecnologías propietarias y sólo disponibles en sistemas del propio Apple, su fuerza está en una base de usuarios con un perfil económico medio-alto, que permite a los desarrolladores fijar unos precios más jugosos y que compensan de sobra la inversión inicial necesaria para entrar en la App Store. Así, actualmente más de la mitad de los beneficios generados mundialmente por la venta de aplicaciones móviles corresponden a aplicaciones diseñadas para el sistema iOS. En el caso de Android, su rápida expansión y disponibilidad en multitud de dispositivos de distintos fabricantes, así como unas tecnologías abiertas que permiten un ciclo de desarrollo de aplicaciones con un coste de inversión casi nulo, le han servido para copar el mercado de dispositivos y aprovechar esta posición de fuerza para atraer a más y mejores desarrolladores. Existe ahora mismo una gran comunidad de desarrolladores escribiendo aplicaciones para extender la funcionalidad de los dispositivos Android. Según datos ofrecidos por Google, la tienda de aplicaciones google play ha sobrepasado ya las 500.000 aplicaciones, sin tener en cuenta aplicaciones de otras tiendas no oficiales (como por ejemplo Samsung Apps, de Samsung, o la AppStore de Amazon). Multitud de aplicaciones están siendo portadas desde el sistema iOS al sistema Android, en busca de una base de usuarios más amplia, y es de esperar que Android se estabilice como sistema operativo dominante de aquí en adelante, igual que lo fue Symbian en su momento.

Sistema Operativo	**Envios 2011 (millones)**	**Cuota de mercado 2011**	**Crecimiento anual**
Android	237.7	48.8%	244%
iOS	93.1	19.1%	96%
Symbian	80.1	16.4%	-29.1%
BlackBerry	51.4	10.5%	5.0%
Bada	13.2	2.7%	183.1%
Windows Phone	6.8	1.4%	-43.3%

Tabla 2. Mercado de teléfonos inteligentes en todo el mundo, según el sistema operativo, en 2011 las ventas mundiales, según Canalys [1]

2.5 El sistema operativo Android

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

Figura 13. "Andy", el robot ideado como logotipo del sistema operativo Android.

2.5.1 Historia

La primera versión de Android fue desarrollada por la compañía Android Inc., una firma fundada en 2003 con el objetivo de crear "(...) dispositivos móviles más inteligentes y que estén más atentos a las preferencias y la ubicación de su propietario". Aunque la evolución del trabajo que se llevaba a cabo en Android Inc. se mantuvo prácticamente en secreto, el gigante Google empezaba a mostrar interés en el negocio de los dispositivos y comunicaciones móviles y en 2005 adquirió la compañía, haciéndola subsidiaria de Google Inc. e incorporando a su plantilla a la mayoría de programadores y directivos originales. A partir de ese momento, Google inició un proceso de búsqueda de acuerdos con diferentes operadores de comunicaciones y fabricantes de dispositivos móviles, presentándoles una plataforma abierta y actualizable y ofreciéndoles diferentes grados de participación en la definición y desarrollo de sus caracterísitcas. La compra por parte de Google de los derechos de varias patentes relacionadas con tecnologías móviles hizo saltar la liebre y los mercados comenzaron a especular con la inminente entrada de la compañía en el negocio de las comunicaciones móviles. Finalmente, el 5 de noviembre de 2007, el sistema operativo Android fue presentado al mundo por la Open Handset Alliance, un recién creado consorcio de más de 70 empresas del mundo de las telecomunicaciones (incluyendo a Broadcom, HTC, Qualcomm, Intel, Motorola, T-Mobile, LG, Texas Instruments, Nvidia, Samsung... por nombrar algunas), liderado por Google y cuyo objetivo es el desarrollo de estándares abiertos para dispositivos móviles. Bajo esta

premisa, Google liberó la mayor parte del código fuente del nuevo sistema operativo Android usando la licencia Apache, una licencia libre, gratuita y de código abierto. La evolución del sistema operativo desde su presentación ha sido espectacular. Sucesivas versiones han ido incorporando funcionalidades como pantalla multi-táctil, reconocimiento facial, control por voz, soporte nativo para VoIP (Voice over Internet Protocol), compatibilidad con las tecnologías web más utilizadas o novedosas como HTML5 y Adobe Flash, soporte de Near Field Communication, posibilidad de convertir el dispositivo en un punto de acceso WiFi, grabación de vídeo en 3D... así como ampliando el espectro de dispositivos soportados, pudiendo encontrar hoy en día Android en el corazón de teléfonos móviles, tablets, ordenadores portátiles y hasta televisores. En la actualidad, los últimos datos indican que Android acapara más del 50% de cuota de mercado de smartphones a escala mundial, con un crecimiento en el último año de un 380% en número de unidades fabricadas; muy por delante de iOS, el sistema operativo del Apple iPhone que, con unacuota del 19%, ha relegado a su vez a Symbian OS a la tercera posición.

2.5.2 Diseño

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

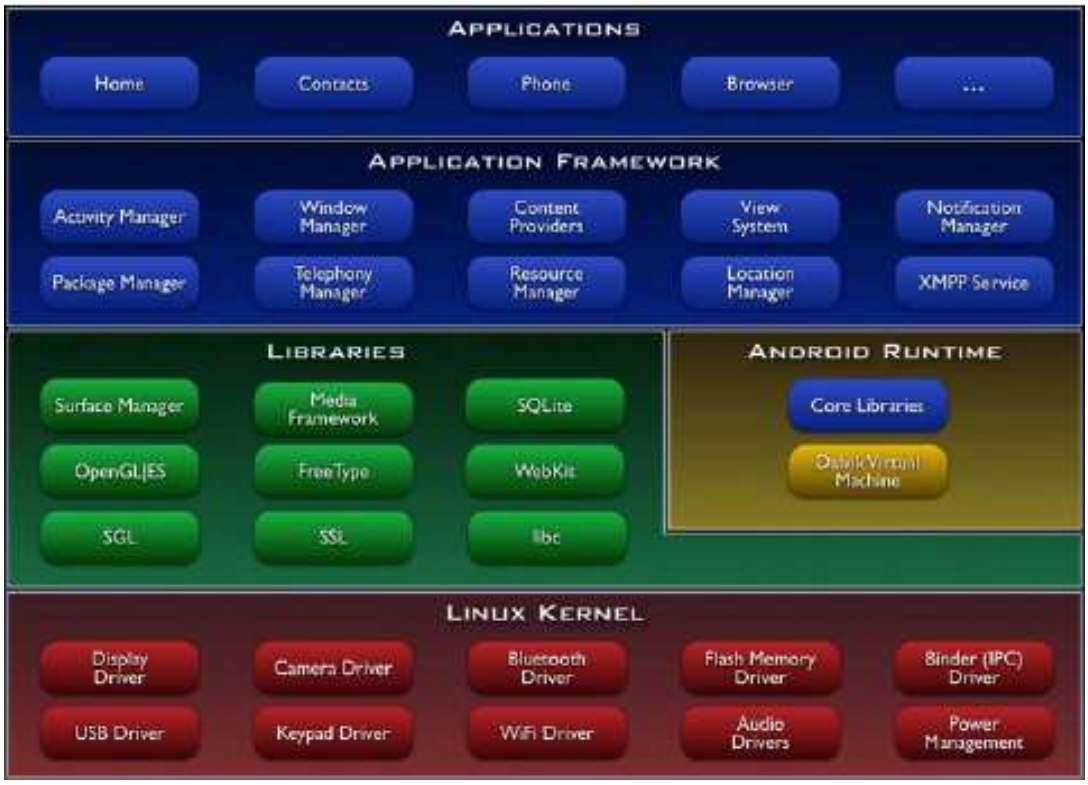

Figura 14. Las diferentes capas que conforman el sistema operativo Android.

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

Applications (aplicaciones): las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros. Todas las aplicaciones están escritas en lenguaje de programación Java.

Application framework (marco de trabajo de aplicaciones): los desarrolladores tienen acceso completo a los mismos APIs del framework usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del framework). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario.

Libraries (bibliotecas): Android incluye un conjunto de bibliotecas de C/C++ usadas por varios componentes del sistema. Estas características se exponen a los desarrolladores a través del framework de aplicaciones de Android. Las bibliotecas escritas en lenguaje C/C++ incluyen un administrador de pantalla táctil (surface manager), un framework OpenCore (para el aprovechamiento de las capacidades multimedia), una base de datos relacional SQLite, una API gráfica OpenGL ES 2.0 3D, un motor de renderizado WebKit,

un motor gráfico SGL, el protocolo de comunicación segura SSL y una biblioteca estándar de C, llamada "Bionic" y desarrollada por Google específicamente para Android a partir de la biblioteca estándar "libc" de BSD.

Android runtime (funcionalidad en tiempo de ejecución): Android incluye un set de bibliotecas base que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java. Cada aplicación Android corre su propio proceso, con su propia instancia de la máquina virtual Dalvik. Dalvik ha sido escrito de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecuta archivos en el formato Dalvik Executable (.dex), el cual está optimizado para memoria mínima. La Máquina Virtual está basada en registros y corre clases compiladas por el compilador de Java que han sido transformadas al formato.dex por la herramienta incluida "dx".

Linux kernel (núcleo Linux): Android dispone de un núcleo basado en Linux para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores, y también actúa como una capa de abstracción entre el hardware y el resto de la pila de software. La versión para Android se encuentra fuera del ciclo normal de desarrollo del kernel Linux, y no incluye funcionalidades como por ejemplo el sistema X Window o soporte para el conjunto completo de librerías GNU, lo cual complica bastante la adaptación de aplicaciones y bibliotecas Linux para su utilización en Android. De igual forma, algunas mejoras llevadas a cabo por Google sobre el núcleo Linux original han sido rechazadas por los responsables de desarrollo de Linux, argumentando que la compañía no tenía intención de dar el soporte necesario a sus propios cambios; no obstante ya se han dado los pasos necesarios para asegurar que todas estas mejoras y controladores se incluyan en las próximas versiones del núcleo oficial.

2.5.3 Estructura básica de una aplicación Android

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

dispositivos Android para instalar nuevas aplicaciones. Estas aplicaciones, una vez instaladas, se ejecutarán como un nuevo proceso en el sistema Linux subyacente, utilizando su propio identificador único de usuario y su propia instancia de la máquina virtual Dalvik. Cuando la aplicación deja de utilizarse o cuando el sistema necesita memoria para otras tareas, el sistema operativo Android finaliza la ejecución del proceso, liberando los recursos. De esta forma, Android crea un entorno bastante seguro en el que cada aplicación, por defecto, sólo puede acceder a los componentes estrictamente necesarios para llevar a cabo su tarea, ya que cada proceso está aislado del resto y no puede acceder a partes del sistema para las cuales no tiene permisos. Como es normal, Android dispone de diferentes maneras de que dos aplicaciones puedan compartir recursos. El más habitual es el sistema de permisos, mediante el cual una aplicación indica qué funcionalidades del sistema necesitará utilizar; la responsabilidad de permitir el acceso recae en el usuario, quien puede aceptar o denegar dichos permisos durante la instalación de la aplicación.

2.5.4 Componentes de una aplicación

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

Actividades (implementadas como subclases de Activity)

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

Servicios (implementados como subclases de Service)

Un servicio no tiene interfaz de usuario, ya que se ejecuta en segundo plano para llevar a cabo operaciones de mucha duración y funcionalidades que, en general, no requieren de intervención del mismo. Siguiendo con nuestro ejemplo de la aplicación musical, ésta

podría implementar un servicio que continúe reproduciendo canciones en segundo plano mientras el usuario lleva a cabo otras tareas, como comprobar su correo o visualizar una presentación de fotografías. La descarga de contenidos desde Internet también se suele llevar a cabo mediante un servicio en segundo plano, que notifica la finalización de la misma a la aplicación original.

Proveedores de contenido (implementados como subclases de ContentProvider)

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

Receptores de notificaciones (implementados como subclases de BroadcastReceiver)

Un receptor de notificaciones responde a notificaciones lanzadas "en abierto" para todo el sistema. Muchas notificaciones las lanza el propio sistema, por ejemplo avisos de batería baja, de que la pantalla se ha apagado o de que hay una llamada entrante. Las aplicaciones también pueden lanzar notificaciones, por ejemplo para informar al resto del sistema de que se ha descargado algún tipo de información y que ya está disponible. Aunque los receptores de notificaciones no tienen interfaz de usuario, sí que pueden informar a éste de los eventos detectados, mediante un icono en la barra de notificaciones. En nuestro ejemplo, el reproductor de música podría tener un receptor de notificaciones para detectar las llamadas entrantes y detener la reproducción en curso o pasar el reproductor a segundo plano.

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

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

2.5.5 Manifiesto de la aplicación

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

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

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

Permisos necesarios: cada aplicación debe informar en su manifiesto de todos los permisos que necesita para su funcionamiento. El usuario no puede aceptar o denegar permisos individuales: o bien los acepta todos e instala la aplicación, o los deniega todos y cancela la instalación.

Librerías y API externas utilizadas: librerías externas a la propia API del sistema Android, que la aplicación utiliza para funcionar; por ejemplo, las librerías de Google Maps.

Requisitos mínimos: en el manifiesto se indican tanto el nivel mínimo de la API de Android obligatorio para el funcionamiento de la aplicación, como características específicas de hardware y software sin las cuales la aplicación no pueda desarrollar sus capacidades. Por ejemplo, si nuestra aplicación utiliza las capacidades multitouch de Android (sensibilidad a múltiples puntos de contacto), tendremos que especificar en el manifiesto que nuestra aplicación requiere una pantalla capaz preparada para multitouch, y al menos un nivel 5 de API (primer nivel en el que se introdujeron los eventos relacionados). El manifiesto de una aplicación se encuentra en el directorio raíz del paquete .apk y, aunque el sistema operativo no utiliza toda la información que contiene, otras aplicaciones como Android Market sí que utilizan esta información para saber si una aplicación es compatible con el dispositivo que está accediendo al servicio.Recursos de la aplicación Además de clases Java y el manifiesto, una aplicación Android se acompaña normalmente de una serie de recursos separados del código fuente, como pueden ser imágenes y sonidos, ficheros XML para las visuales y cualquier otro recurso relacionado con la presentación visual o la configuración de la aplicación. Estos recursos se organizan en directorios dentro del paquete .apk como sigue (se muestran sólo los más habituales):

res: directorio raíz de los recursos.

drawable: este directorio contiene imágenes en formato PNG, JPG o GIF, que después se cargarán en la aplicación como objetos de tipo Drawable.

layout: directorio para almacenar las visuales XML de la aplicación, es decir, ficheros XML que contienen la definición de las interfaces de usuario.

menu: contiene ficheros XML que definen los menús utilizados en la aplicación.

values: agrupa ficheros XML que definen múltiples recursos de tipos sencillos, como cadenas de texto predefinidas, valores enteros constantes, colores y estilos. Para definir cadenas se suele utilizar el fichero strings.xml, para definir estilos el fichero styles.xml, etc.

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

Para cada uno de estos recursos (excepto los contenidos del directorio assets), el SDK de Android genera un identificador único y lo almacena como una constante dentro de la

clase R.java. Así, una imagen situada en res/drawable/thumb1.png podrá referenciarse mediante la variable R.drawable.thumb1, y R.layout.principal apuntaría a una visual XML guardada en res/layout/principal.xml.

Además, el desarrollador puede añadir sufijos a cualquiera de los directorios para definir recursos que sólo estarán disponibles cuando se den las condiciones definidas por ese sufijo. Un uso muy habitual es definir directorios drawable-ldpi y drawable-hdpi para almacenar recursos que sólo deben utilizarse en casos de que el dispositivo tenga una resolución de pantalla baja (ldpi) o alta (hdpi). Otro uso es disponer de versiones "localizadas" de las distintas cadenas de texto utilizadas en la aplicación, creando directorios como res/values-en y res/values-de para guardar la traducción al inglés y al alemán de los textos de nuestra aplicación. Con esto finalizamos la introducción al sistema operativo Android y sus aplicaciones. Para conocerlo en mayor profundidad, se puede acudir a la bibliografía, aunque es más edificante construir nuestra propia aplicación Android y aprender sobre la marcha.

2.6 **Patrones de diseño**

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

2.6.1 El patrón MVC (Model-View-Controller)[7]

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

A continuación se presenta una explicación algo más detallada de cada componente:

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

Vista: recoge la información del Modelo y la presenta al usuario en una forma tal que pueda ser interpretada y manejada por el mismo, casi siempre a través de una interfaz gráfica; engloba también toda la lógica de interactuación con los dispositivos físicos que conforman dicha interfaz de usuario (pantallas, teclados, superficies sensibles al tacto), incluyendo la generación de los eventos necesarios, que serán recibidos por el Controlador. Gracias al bajo acoplamiento con el Modelo, una misma aplicación puede tener diferentes Vistas intercambiables entre sí para presentar la información de diferentes maneras.

Controlador: recibe los eventos generados por el usuario a través de la Vista, los procesa y genera las respuestas adecuadas en el Modelo. En ocasiones la acción del usuario no provoca una actuación sobre el Modelo, sino sobre la propia Vista; en tal caso el Controlador se encarga también de iniciar las rutinas necesarias para que la Vista refleje estos cambios.

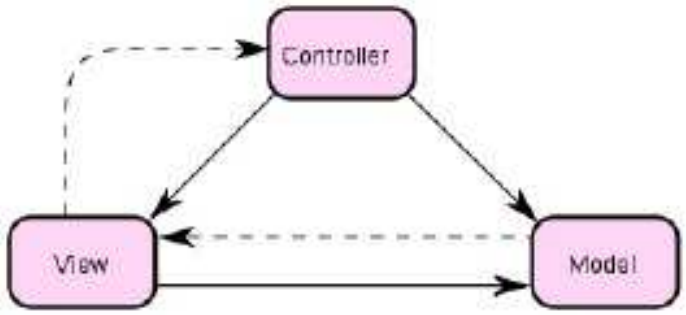

Figura 15. Diagrama ideal de relación entre los componentes del patrón MVC. La Vista tiene conocimiento del Modelo, y el Controlador de ambos, pero no a la inversa.

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

- El usuario lleva a cabo algún tipo de acción sobre la interfaz de usuario; esto es, sobre la Vista. Por ejemplo, pulsar un botón.

- El Controlador, normalmente tras registrarse como receptor, recibe de la Vista el evento generado por la acción del usuario y lo gestiona.

- En caso necesario (si no es una simple consulta), el Controlador accede al Modelo para actualizar su estado, por ejemplo eliminando un registro; o simplemente notifica al Modelo la operación a realizar, en el caso de que los objetos del Modelo incluyan su propia lógica de negocio interna.

- El Modelo actualiza su estado.

- La Vista actualiza la interfaz de usuario para reflejar los cambios llevados a cabo en el Modelo.

- La aplicación queda a la espera de la siguiente acción del usuario.

2.6.2 Aproximaciones al patrón MVC

En las implementaciones de un patrón MVC puro, Controlador y Vista deben mantener un alto grado de desacoplamiento. El Controlador no interviene en el proceso de actualización de la Vista, dejando que sea ella misma la que detecte los cambios en el Modelo, mediante eventos o por consulta directa. De igual forma, la Vista no incluye la lógica para gestionar los eventos del usuario; éstos son recibidos directamente por el Controlador, que es quien decide la actuación que se debe llevar a cabo en base a cada evento. No obstante, mantener este grado de desacoplamiento no siempre es fácil y, en ocasiones, tampoco es productivo puesto que fija a los desarrolladores de cada componente unos límites a veces artificiales. Como respuesta a estas situaciones, basándose en el patrón MVC original se han ideado múltiples patrones derivados que trasladan hacia un lado (Vista) u otro (Controlador) el peso de la lógica principal de la aplicación, según convenga al escenario y tipo de aplicación concretos. Por ejemplo, hay escenarios en los que interesa centralizar en la Vista todas las acciones relacionadas con la interfaz de usuario, siendo la propia Vista la receptora de los eventos generados por el usuario, no el Controlador; mientras que el Controlador se encarga del tratamiento de los datos y de adaptar el formato de los mismos de manera que puedan ser procesados fácilmente por la Vista, pero sin conocer directamente las acciones que el usuario genera, sólo aquellas que la Vista delega en él. Esta variante del patrón MVC es conocida como "Modelo-Vista-Presentador" o MVP (Model-View-Presenter), y fue derivada directamente de la implementación que incluía Smalltalk de MVC.

Su objetivo es la simplificación del modelo en dos partes correspondientes a las dos cuestiones que se plantean durante el desarrollo de una aplicación: por un lado, cómo maneja el programa los datos, de lo cual se encarga la lógica de comunicación entre el Controlador y el Modelo; y por el otro, cómo interactúa el usuario con dichos datos, cuestión que resuelve la Presentación, es decir la combinación de la Vista y el Presentador (o Controlador).

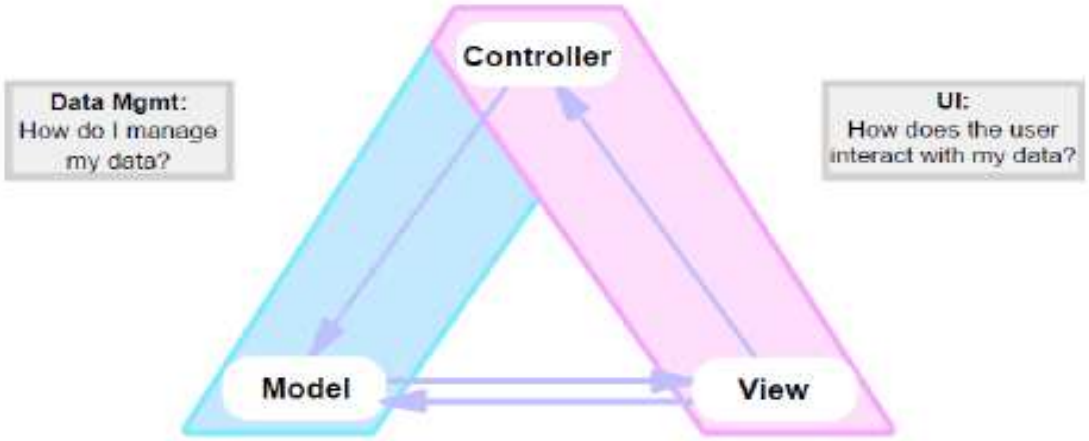

Figura 16. Ilustración de la distinción que el patrón MVP (Modelo-Vista-Presentador) hace entre Modelo y Presentación.

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

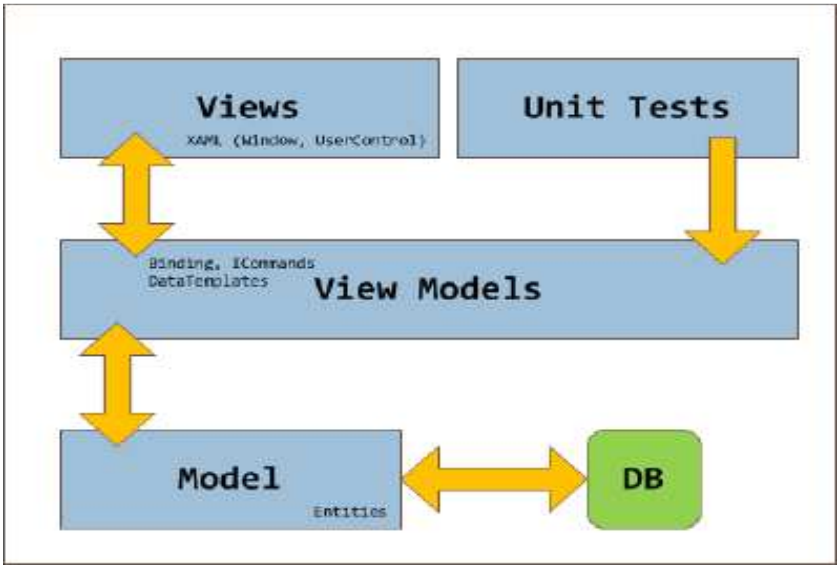

Figura 17. Relación entre los distintos componentes del patrón MVVM: Modelo, Modelo de la Vista, y Vista.

La ventaja de este patrón es que elimina de la Vista todo el código "de respaldo", es decir, código necesario para el manejo de los datos pero que no cumple una función visual determinada. De esta manera se facilita la división de trabajo en aplicaciones con Vistas interfaces de usuario elaboradas, que pueden ser desarrolladas por profesionales especializados sin necesidad de conocer el Modelo, mientras que por otro lado desarrolladores más enfocados a la programación tradicional pueden encargarse de la lógica que maneja los eventos de usuario y el proceso de datos. Además se facilita el testeo de la parte visual de la aplicación, al estar más centralizada toda la funcionalidad.

Estas variantes del patrón MVC se adaptan mejor a la arquitectura general de una aplicación Android que un patrón MVC puro, como veremos más adelante, debido a que el alto acoplamiento que existe entre las clases Activity y la clase o clases SurfaceView asociadas a las mismas propicia el trasvase de competencias entre el Controlador (la clase Activity y auxiliares) y la Vista (las clases SurfaceView y sus visuales definidas en XML). Esto no quiere decir que los conceptos relativos al patrón MVC no sean aplicables a nuestro programa. Aunque no pueda considerarse un patrón MVC puro, muy pocas aplicaciones del mundo real consiguen una implementación exacta del patrón MVC debido a la dificultad de lograr un desacoplamiento total entre los tres componentes, y la variante utilizada a lo largo de este proyecto representa un didáctico ejemplo de aproximación al modelo ideal MVC en una aplicación real.

3 Entorno de desarrollo

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

3.1 Android SDK

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

Figura 18. Aspecto del emulador de dispositivo Android incluido con el SDK.

Algunas de las herramientas más importantes son:

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

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

3.2 **Eclipse**

Eclipse es una plataforma de desarrollo abierta y libre, extensible mediante módulos y plug-ins. Está orientada inicialmente a la creación de aplicaciones Java mediante sus funcionalidades como entorno de desarrollo integrado (IDE, por sus siglas en inglés); sin embargo, su código abierto, sus posibilidades de extensión y la posibilidad de crear aplicaciones completas utilizando la propia plataforma Eclipse como base, han hecho que se cree una gran comunidad a su alrededor que proporciona, además de compatibilidad con múltiples lenguajes adicionales (desde Ada hasta Python pasando por C++, Scheme, Perl o COBOL), módulos de extensión y herramientas para cualquier tarea que se nos presente a lo largo del ciclo de vida de nuestra aplicación. ¿Necesitas control de versiones? Eclipse permite de forma nativa trabajar con repositorios CVS y, mediante complementos, con otros sistemas más avanzados como Subversion, Mercurial o git.

¿Estás desarrollando una aplicación web con JSP y servlets? Eclipse descarga automáticamente plug-ins para los contenedores más conocidos (Tomcat, WebLogic, etc.) y gestiona tanto el arranque y parada del servidor como todo el proceso de empaquetado, publicación y despliegue de la aplicación web. ¿Depuración de código Java? Eclipse ofrece múltiples opciones de ejecución en modo depuración, puntos de interrupción, seguimiento de variables e incluso sustitución de código "en caliente" (si la máquina virtual lo permite). ¿Diagramas UML? El sitio web de Eclipse tiene secciones enteras dedicadas a herramientas y módulos enfocados al "desarrollo por modelos".

Eclipse surgió como un proyecto de la división canadiense de IBM para sustituir su línea de productos VisualAge, desarrollados en SmallTalk, por una plataforma más moderna basada en Java. En 2001 se decidió liberar el código del proyecto, creándose un consorcio de más de 50 empresas que apoyó el desarrollo abierto de la plataforma. La Fundación Eclipse se creó a partir del consorcio original y actualmente está compuesta por más de 180 miembros, como por ejemplo: Adobe, AMD, Borland, Cisco, Google, HP, IBM, Intel, Motorola, Nokia, Novell, Oracle, SAP o Siemens, por citar algunos de los más conocidos. Actualmente, Eclipse es una plataforma ampliamente usada (quizá la que más) tanto por desarrolladores independientes como en entornos empresariales, dado que la mayoría de herramientas, módulos y plug-ins existentes para la misma son libres y gratuitos y representa, por tanto, la mejor opción calidad-precio a la hora de elegir un entorno de desarrollo potente y fiable, teniendo un coste prácticamente nulo.

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

3.3 **ADT para Eclipse**

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

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

3.4 **Ejemplo de desarrollo**

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

Este ejemplo está basado en el tutorial Hello, World que se encuentra en la página web oficial de Android para desarrolladores, referenciada en la bibliografía. El detalle de cada uno de los pasos se puede consultar en dicho tutorial, ya sea de manera online, o en la versión offline que se descarga junto con la documentación del SDK de Android.

Este ejemplo supone así mismo que tenemos instaladas y configuradas todas las herramientas anteriores, y que ya hemos descargado al menos una versión de Android sobre la cual desarrollar nuestro ejemplo. Si no es así, se recomienda consultar la bibliografía para ampliar la información acerca de cómo hacerlo.

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

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

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

Construir la visual XML. Un proyecto Android puede tener tantas visuales XML como nosotros queramos, todas ellas alojadas en el directorio res/layout del proyecto; la visual por defecto se llama main.xml y se crea al mismo tiempo que el proyecto. Podemos

desecharla si queremos, pero en nuestro caso es más sencillo editarla para dejarla de la siguiente manera:

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

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

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

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

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

Ejecutar la aplicación. Ahora ya podemos ejecutar la aplicación y observar el resultado. Para ello sólo tenemos que pulsar Run as... Android application en el menú contextual del proyecto. Si hemos definido bien el dispositivo Android virtual, Eclipse lo seleccionará

automáticamente para ejecutar la aplicación, o bien nos pedirá que seleccionemos nosotros uno de los disponibles. El resultado debe ser algo similar a ésto:

Figura 19. Aplicación de ejemplo en ejecución.

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

4 La aplicación: NAVES

Como se ha descrito en la introducción del proyecto, esta aplicación dispone de dos modos de funcionamiento, automático e interactivo, los cuales se han hecho como aplicaciones separadas y un tercer modo de funcionamiento a través de una red wifi.

A lo largo de este capítulo se describirá la aplicación NAVES en modo automático, tanto a nivel funcional general como entrando en el detalle técnico siempre que se considere necesario o de interés, después se pasa a explicar el modo manual, se expondrán los puntos que difieren del modo automático, y para finalizar se explica el modo de juego en red.

Comenzaremos explicando los distintos modos de funcionamiento y un pequeño manual de instalación y ejecución, la estructura general de la aplicación, sus actividades, identificando la correspondencia entre cada una de las partes del patrón MVC (Modelo, Vista y Controlador) y dichas clases. A continuación iremos viendo cada una de las actividades que conforman la aplicación, sus pantallas y utilización, y las clases y recursos relacionados. Por último se hará un análisis de las fortalezas y carencias de la aplicación y mejoras que podrían aplicarse a nivel técnico.

4.1 La aplicación.

4.1.1 Modos de funcionamiento.

Como ya se a expuesto antes esta aplicación tiene tres modos de funcionamientos, que se distribuyen por separado como tres archivos instalables. Cada uno de los modos de funcionamiento servirá además para explicar alguna característica o conjunto de características de la aplicación. De esta manera:

- El modo de funcionamiento automático se ha hecho para explicar como un juego actualiza los datos y redibuja la pantalla, sin la intervención del usuario, y como funciona Android con respecto a las vistas XML, captura de eventos en la interface visual y demás características sobre la programación en Android.

- El modo de funcionamiento manual se utiliza para explicar la utilización de sensores, como el acelerómetro.

- El modo de funcionamiento en red se utiliza para explicar la utilización de socket, en concreto socket UDP, en Android. Esta versión tiene un servidor que se ejecuta en un PC y un cliente que se ejecuta en el dispositivo Android.

4.1.2 Descripción del juego

A continuación se realizará una pequeña descripción del juego en cada modo.

- Modo automático: Este modo el juego es totalmente autónomo, el usuario añade naves, “Patrulla” ó “Patera”, que se mueven por la pantalla, y cuando una nave Patrulla detecta una Patera la persigue y la nave Patera intentará huir.

- Modo Manual: En este modo el usuario añade naves al juego, y al mover el dispositivo la naves se moverán en el sentido del movimiento del dispositivo, si una nave Patera entra en el radar de una nave Patrulla, esta automáticamente irá hacia la nave patera.

- Modo red: Una vez el servidor activo y habiendo varios clientes conectados, estos se moverán sobre toda la superficie de juego, si una nave entra en el radar de otra, se vera reflejado en el dispositivo cliente.

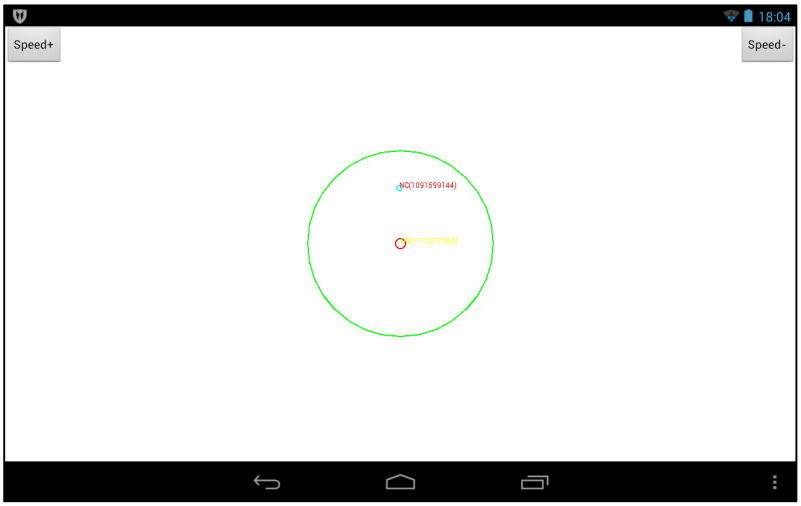

Figura 20 Pantalla radar muestra nave localizada

4.1.3 Mini manual de instalación y ejecución

A continuación se explicará como instalar y ejecutar cada modo de funcionamiento.

Para instalar las distintas versiones en el dispositivo Android, se ejecuta el archivo *apk*(1) de cada modo en el dispositivo Android.

Los modos automático y manual se ejecutan en el dispositivo Android, sin necesidad de sistemas externos.

El modo en red se compone de dos partes: la primera, el servidor, que se ejecuta en un PC y se proporciona como un archivo *.jar*(2), la segunda, el cliente, que se instala en un dispositivo android y se conecta a través de wifi al servidor. Para un correcto funcionamiento de este modo, hay que ejecutar primero el servidor en el PC y después se ejecuta en el dispositivo Android.

(1)Un archivo .apk es un instalador de programa android, similar al setup.exe de Windows.
(2)Un archivo .jar es un ejecutable de un programa java.

4.1.4 Ciclo de vida de la aplicación

A continuación de muestra un diagrama del ciclo de vida de la aplicación.

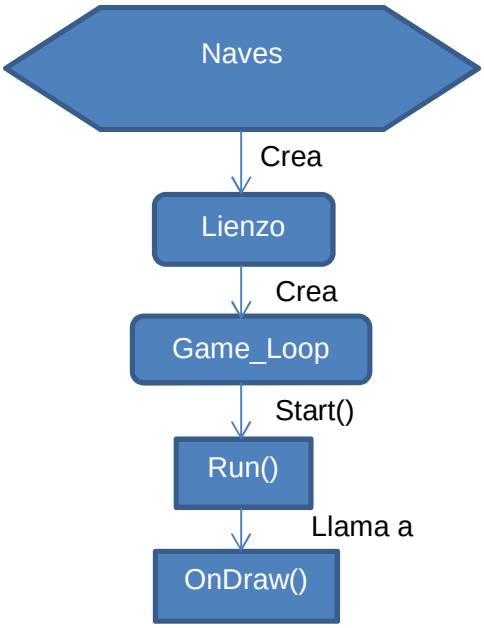

Figura 21. Diagrama de ciclo de vida de la aplicación

El método *run()* llama cada cierto tiempo al método *OnDraw(),* el tiempo de llama se mide en milisegundos.

4.2 Aplicación NAVES modo automático

4.2.1 Estructura general de la aplicación

La aplicación dispone de una sola actividad

`Naves_automatico`: Contiene el código necesario para que el usuario pueda interactuar con la aplicación y ejecuta otras clases para el funcionamiento normal de la aplicación, como es la clase `Lienzo`.

Las dos clases más importantes son:

- **Clase `Lienzo[2]`**: Es la clase mas importante, pues es la que dibuja el pantalla las naves, es una extensión de la clase Android `SurfaceView`, la cual es utilizada para dibujar en pantalla objetos en movimiento, esta clase la explicaremos con mas detalle mas adelante.
- **Clase `GameLoop[2]`**: Esta clase proporciona la velocidad de refresco al movimiento de los objetos (naves) de la aplicación.

4.2.2 Organización clases y recursos

Como es normal y recomendable en cualquier proyecto Java, las clases de la aplicación NAVES se han organizado en paquetes para una mejor identificación de sus roles. Como nombre "base" de la paqueterización se ha elegido proyecto.naves_auto y a partir de dicho paquete raíz se han ido creando diferentes subpaquetes atendiendo a las características de las clases. Para esta división en subpaquetes se podría haber elegido uno cualquiera de entre múltiples criterios, como por ejemplo el perfil de las clases dentro del patrón MVC: modelos, controladores y vistas; su relación con la API de Android: activities, views, adapters, overlays; o la funcionalidad a la que van asociadas: Dibujado en pantalla, actualizar datos. El ordenamiento elegido diferencia Interfaces de clases

. La paquetización queda como sigue:

proyecto.naves_auto
Contiene las clases que componen la aplicación controladores, vistas y modelos.

proyecto.naves_auto.interfaces
Contiene las interfaces que definen el comportamiento de las naves.

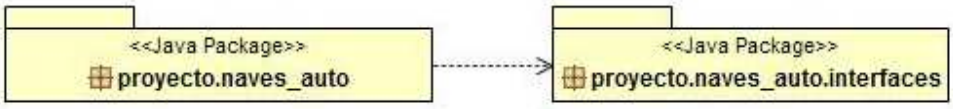

Figura 22. Diagrama de paquetes de la aplicación NAVES, con las relaciones de dependencia entre los mismos.

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

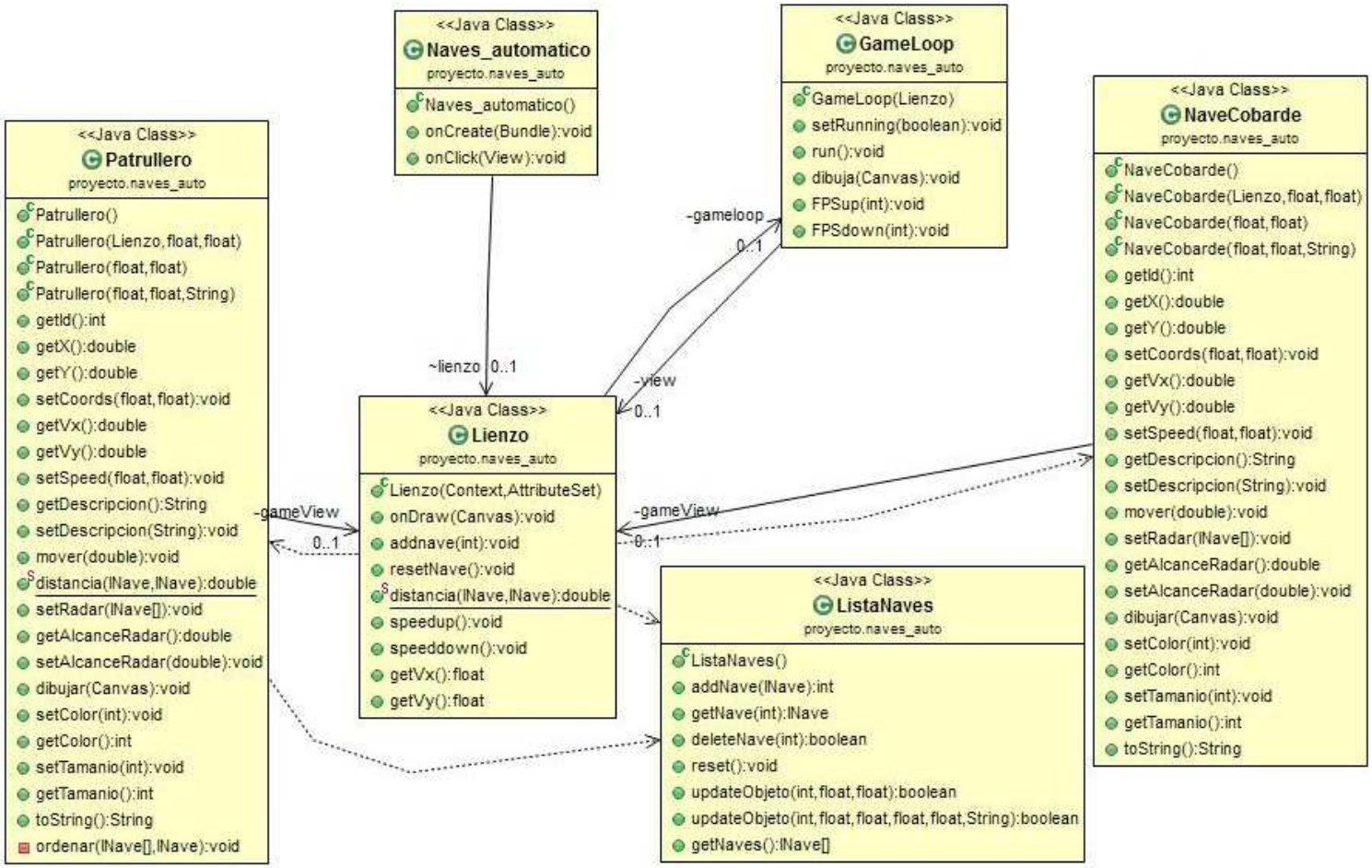

Figura 23. Diagrama de clases de la aplicación NAVES automático

4.2.3 Diseñando el Modelo: `Patrullero`, `NaveCobarde`

Esta es la representación específica de la información con la cual el sistema opera. En resumen, el modelo se limita a lo relativo de la *vista* y su *controlador* facilitando las presentaciones visuales complejas.

En esta aplicación los modelos definen el comportamiento de los objetos del juego, y estos objetos son los distintos tipos de naves. Las cuales vemos en mas detalle en los siguientes apartados.

4.2.4 Interfaces `INave`, `IDibujable`, `IObservadoresNaves`

Esta interfaces definen las funciones para que las clases *`Patrullero`* y *`NaveCobarde`* obtengan el comportamiento de una nave, ambas clases implementan las tres interfaces.

La interface *`INave`* define las funciones de movimiento e identidad (cambiar y recuperar posición, velocidad, id y descripción).

```
public interface INave {
	/*
	 * Obtiene la id de la nave
	 * @return devuelve la id de la nave de tipo int
	 */
	public int getId();
	/*
	 * obtiene la coordenada X de la nave
	 * @return devuelve la coordenada X de tipo double
	 */
	public double getX();
	/*
	 * obtiene la coordenada Y de la nave
	 * @return devuelve la coordenada Y de tipo double
	 */
	public double getY();
	/*
	 * Define las coordenadas X e y
	 * @param x coordenada X del plano de tipo float
	 * @param y coordenada y del plano de tipo float
	 */
	public void setCoords(float x, float y);
	/*
	 * obtiene la velocidad X de la nave
	 * @return devuelve la velocidad X de tipo double
	 */
	public double getVx();
	/*
	 * obtiene la velocidad Y de la nave
	 * @return devuelve la velocidad Y de tipo double
	 */
```

```
        public double getVy();
        /*
         * Define las velocidad X e y
         * @param x velocidad X del plano de tipo float
         * @param y velocidad y del plano de tipo float
         */
        public void setSpeed(float vx, float vy);
        /*
         * Obtiene una descripción de la nave
         */
        public String getDescripcion();
        /*
         * Define una descripción de la nave
         */
        public void setDescripcion(String descripcion);
        /*
         * Este método actualiza los datos de movimiento de la nave
         * @param delta_t incremento de velocidad de tipo double
         */
        public void mover(double delta_t);
}
```

La interface *IDibujable* define las funciones para el dibujado en pantalla:

```
public interface IDibujable {
        /*Metodo para dibujar la nave en pantalla
         *
         * @param g parametro de la clase Canvas, superficie de dibujo
         */
        public void dibujar(Canvas g);
        /*
         * Define el color de la nave
         * @param c numero de color
         */
        public void setColor(int c);
        /*
         * Devuelve el color de la nave
         * @return numero de color
         */
        public int getColor();
        /*
         * Define el radio de la nave
         * @param tam tamaño del radio
         */
        public void setTamanio(int tam);
        /*
         * Recupera el tamaño de la nave
         * @return Devuelve el radio de la nave
         */
        public int getTamanio();
}
```

La interface *IObservadoresNaves* define las funciones para el manejo del radar de la nave:

```
public interface IObservadoresNaves {
	/*
	 * Define y rellena la lista de naves a perseguir
	 * @param lista array de INaves
	 */
	public void setRadar(INave [] lista);
	/*
	 * obtien el alcance del radar de la nave
	 * @return Devuelve el radio del radar
	 */
	public double getAlcanceRadar();
	/*
	 * Define el tamaño del radar
	 * @param alcance es el tamaño del radio del radar de la nave
	 */
	public void setAlcanceRadar(double alcance);
}
```

4.2.5 Clase `Patrullero`

Esta clase modela el comportamiento del tipo de nave 'patrulla', los atributos necesarios son:

Variables	Descripción
`vx y vy`	Velocidad sobre el eje 'x' y el eje 'y' respectivamente (tipo float).
`descripción`	Describe el tipo de nave (tipo String).
`id`	Numero identificativo único de cada nave (tipo int).
`color`	Identifica el color con el cual se va a dibujar en pantalla (tipo int).
`Tamanio`	Establece en tamaño del circulo que representa la nave (tipo int).
`'x' e 'y'`	Coordenadas 'x' e 'y' donde se dibuja la nave (tipo float).
`alcance`	Establece el tamaño del radar de la nave (tipo double).
`[] miLista`	En este array se almacenan las naves que están dentro del radar (tipo INave).

`Inspeccionados`	Es una lista donde se almacenan las naves inspeccionadas (tipo IListaNaves).
`target`	Es la nave que persigue la patrulla (tipo INave).
`countInspeccion`	Contador para añadir una nave a la lista de inspeccionados (tipo int).
`gameView`	Instancia de la clase Lienzo, es necesaria para diversas operaciones.

El constructor utilizado necesita una instancia de Lienzo para que la nave sea dibujada en el.

Los métodos más destacables son:

Métodos	Descripción
`mover(double delta_t)`	Este método contiene la algoritmia necesaria para la actualización de la posición de la nave sobre el lienzo, en el caso de el radar detecte naves o no, si el radar detecta una nave tipo `NaveCobarde` este la perseguirá.
`dibujar(Canvas g)`	La única función de este método es dibujar la nave en pantalla.

4.2.6 Clase `NaveCobarde`

Esta clase modela el comportamiento del tipo de nave 'patera', los atributos necesarios son:

Variables	Descripción
`vx y vy`	Velocidad sobre el eje 'x' y el eje 'y' respectivamente (tipo float).
`descripción`	Describe el tipo de nave (tipo String).
`id`	Numero identificativo único de cada nave (tipo int).
`color`	Identifica el color con el cual se va a dibujar en pantalla (tipo int).
`Tamanio`	Establece en tamaño del círculo que representa la nave (tipo int).

`'x' e 'y'`	Coordenadas 'x' e 'y' donde se dibuja la nave (tipo float).
`alcance`	Establece el tamaño del radar de la nave (tipo double).
`[] miLista`	En este array se almacenan las naves que están dentro del radar (tipo INave).
`Inspeccionados`	Es una lista donde se almacenan las naves inspeccionadas (tipo IListaNaves).
`huyendo`	Identifica si la nave esta huyendo de una patrulla, para cambiar su trayectoria (tipo boolean).
`gameView`	Instancia de la clase Lienzo, es necesaria para diversas operaciones.

El constructor utilizado necesita una instancia de Lienzo para que la nave sea dibujada en el.

Los métodos más destacables son:

Métodos	Descripción
`mover(double delta_t)`	Este método contiene la algoritmia necesaria para la actualización de la posición de la nave sobre el lienzo, en el caso de el radar detecte naves o no, si el radar detecta una nave tipo `Patrulla` este huirá.
`dibujar(Canvas g)`	La única función de este método es dibujar la nave en pantalla.

4.2.7 Controladores y Vistas

Con lo que hemos visto en los capítulos correspondientes al sistema operativo Android y al patrón MVC, sabemos que la identificación de Controlador y Vista en una aplicación Android no es tan clara como en otros enfoques. Las visuales XML, aunque definen completamente la visual, siguen necesitando de las clases Java correspondientes para instanciar cada uno de los elementos declarados, con sus propiedades y atributos correspondientes. Las clases Java de tipo *View*, por otra parte, en varias ocasiones no se limitan estrictamente a ofrecer una interfaz visual para el usuario, sino que añaden lógica auxiliar para facilitar el control de las mismas por parte del programador, a costa de

traspasar la línea que separa la Vista del Controlador. En cuanto a las clases derivadas de *Activity*, según cómo se utilicen pueden asumir funciones propias de la capa Visual, añadiendo nuevos elementos a la interfaz de usuario según las necesidades de la aplicación y tomando el rol de ViewModel ya explicado. Para definir un criterio único, en adelante identificaremos siempre como el Controlador del patrón MVC a las Actividades (clases derivadas de *Activity*), así como a cualquier otra clase auxiliar que no sea de tipo View y que ejecute lógica necesaria para llevar a cabo las funcionalidades de la aplicación. La Vista se corresponderá, por tanto, con las visuales XML y las clases derivadas de View, además de aquellas clases que sean utilizadas por éstas y cuya lógica se limite a actuar sobre la interfaz de usuario directa o indirectamente, sin llevar a cabo otro tipo de tareas como gestión de eventos o del ciclo de vida de la aplicación.

Así, las clases `Naves_automatico` y `GameLoop` serían controladores y las clases `Lienzo`, extendida de `SurfaceView`, y `ListaNaves`, la cual se encarga de almacenar las naves para dibujarlas, serian parte de la vista dado que son las encargadas de dibujar los gráficos en pantalla, directa e indirectamente respectivamente.

Con estos criterios podemos analizar más en profundidad cada una de las clases comentadas anteriormente.

4.2.8 Clase `Naves_Automatico`

Esta es la clase principal de la aplicación, esta extendida de la clase *Activity*, es la encargada de recoger los eventos de pantalla, da interactividad con el usuario, y a su vez muestra la interface grafica en pantalla.

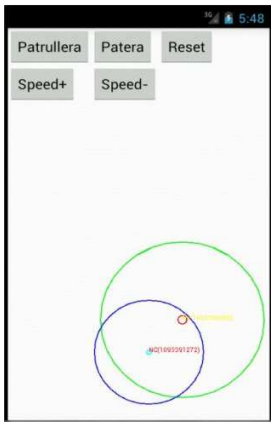

Figura 24. Interface grafica de la Aplicación NAVES

Esta clase tiene dos partes, una es la instanciación de los botones y superficie de dibujo y la otra el manejo de eventos.

4.2.9 Clase `GameLoop`

Esta clase marca en tiempo de refresco de la imagen, para dar efecto de moviendo a las naves.

Es una extensión de la clase *Thread,* crea un hilo aparte y llama cada cierto tiempo a al método *onDraw(Canvas c)* para redibujar la superficie de dibujo.

El método mas destacable de esta clase es el método *run()*.

```
public void run() {
         handler.removeCallbacks(rutina);
         handler.postDelayed(rutina, FPS);
       }

       private Runnable rutina = new Runnable() {
           public void run() {
                Canvas c = null;
                dibuja(c);
              handler.removeCallbacks(rutina);
              handler.postDelayed(this, FPS);
           }
         };

       public void dibuja(Canvas c){
          try {
                c = view.getHolder().lockCanvas();
                synchronized (view.getHolder()) {
                       view.onDraw(c);
                }
        } finally {
                if (c != null) {
                       view.getHolder().unlockCanvasAndPost(c);
                }
        }
       }
```

Esta clase funciona de la siguiente manera:

Se crea una instancia de la clase *`Handler` (manejador)*, esta clase crea una cola de "tareas", estos "tareas" son llamadas a una subrutina *`runnable`*.

La llamada a `handler.removeCallbacks(rutina)` saca de la cola la tarea `rutina` y al llamar a `handler.postDelayed(rutina, FPS)` se vuelve a meter la tarea de llamar a `rutina` pero se ejecuta pasados `FPS` milisegundos

En el método *`dibuja`* bloquea el canvas (superficie de dibujo) para que mientras esta clase este modificándolo nadie pueda usarlo, y se crea un bloque *`synchronized`* para

asegurarnos de que solo esta clase llama al método de dibujado en pantalla, una vez redibujada la pantalla se desbloquea el canvas.

4.2.10 Clase `Lienzo`

Esta clase es extendida de la clase *SurfaceView*, la cual proporciona una superficie de dibujo, para controlar esa superficie de dibujo se utiliza la clase *SurfaceHolder* que da acceso a los métodos para crear, destruir o realizar cambios en la superficie de dibujo.

El código de lo explicado queda de la siguiente manera:

```
public Lienzo(Context context, AttributeSet attributeSet) {
        super(context, attributeSet);
        gameloop = new GameLoop(this);
         holder = getHolder();
         holder.addCallback(new SurfaceHolder.Callback() {

            //Destruye la superficie de dibujo
                public void surfaceDestroyed(SurfaceHolder holder) {
                   boolean retry = true;
                    gameloop.setRunning(false);
                    while (retry) {
                           try {
                                  gameloop.join();
                                  retry = false;
                           } catch (InterruptedException e) {
                           }
                    }
                }
                  //Crea la superficie de dibujo
                public void surfaceCreated(SurfaceHolder holder) {
                    gameloop.setRunning(true);
                     gameloop.start();
                }

                public void surfaceChanged(SurfaceHolder holder, int format,
int width, int height) {
                }
          });
      }
```

Para realizar los dibujos en la superficie se utiliza método *`onDraw(Canvas canvas)`*, este método lo llama la clase *`GameLoop`* para redibujar la superficie y actualizar los datos de movimiento.

```
public void onDraw(Canvas canvas) {
       if(canvas!=null){
             canvas.drawColor(Color.BLACK);
             naves=listaNaves.getNaves();

             if(naves!=null){
                   for (int i = 0; i < naves.length; i++) {
                         if (naves[i] instanceof IDibujable){
                               ((IDibujable)naves[i]).dibujar(canvas);
                          }
                         if (naves[i] instanceof IObservadoresNaves){
                               IListaNaves lista = new ListaNaves();
                               for(int j = 0; j < naves.length; j++){
                                     if (j != i && distancia(naves[i],
naves[j]) < ((IObservadoresNaves)naves[i]).getAlcanceRadar()){
                                           lista.addNave(naves[j]);
                                     }
                               }

      ((IObservadoresNaves)naves[i]).setRadar(lista.getNaves());
                         }
                    }
             }
       }
    }
```

4.2.11 Interface `IListaNaves`

La interface *IListaNaves* define las funciones para el control de las naves de la apliciones:

```
public interface IListaNaves {

      public static final int MaxNaves = 100;
/*
 * Añade una nueva nave a la lista
 * @param obj del tipo INave
 * @return devuelve el numero de ID de la nave añadida
 */
      public int addNave(INave obj);
      /*
       * borra una nave
       * @param id numero de identificación de la nave a borrar
       * @return Devuelve true si se a borrado bien y false si no se ha borrado
o la id no existe
       */
      public boolean deleteNave(int id);
      /*
       * Busca una nave en concreto dentro de la lista
```

```
        * @param id Identificación de la nave a buscar
        * @return devuelve la nave si existe y si no devuelve null
        */
       public INave getNave(int id);
       /*
        * lista las naves existentes
        * @return Devuelve un array de INaves existentes
        */
       public INave [] getNaves();
       /*
        * Borra todas las naves de la lista
        */
       public void reset();
}
```

4.2.12 Clase `ListaNaves`

Esta clase implementa la *interface* *`IListaNaves`*. Es auxiliar de la clase *`Lienzo`*, su función es simple, llevar el control de las naves existentes en la aplicación, esta clase añade, recupera y borra una nave, borra y recupera todas las naves existentes.

4.3 **Aplicación NAVES modo manual**

Para este modo, el control de las naves será a través del acelerómetro, para ello se crea una nueva clase llamada `Acelerometro` y se adapta la clase `Lienzo` para la actualización de los datos de velocidad a través de los datos recuperados del sensor de aceleración del dispositivo.

El diagrama de clases queda de la siguiente manera:

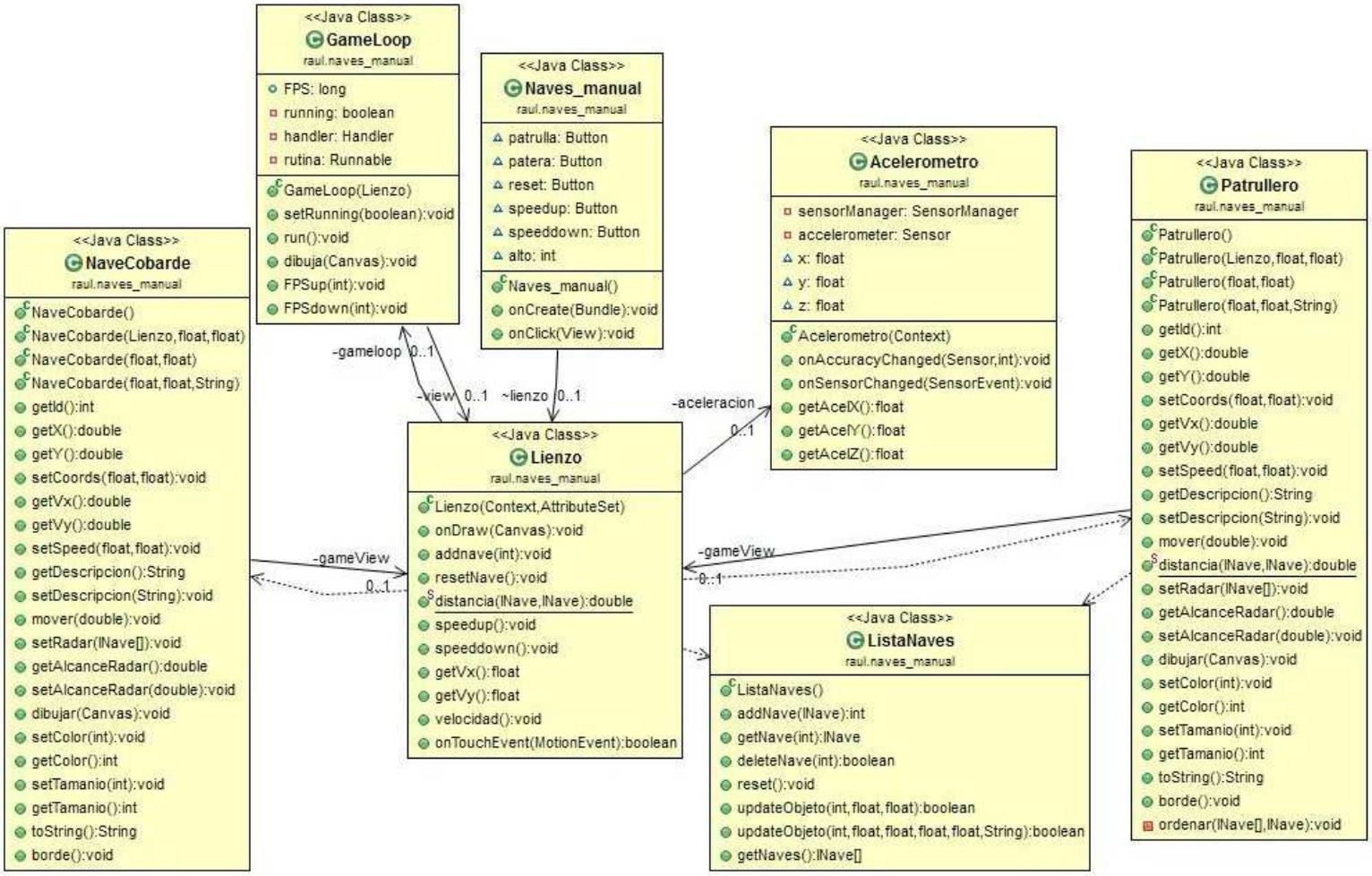

Figura 25. Diagrama de clases de la aplicación NAVES manual

4.3.1 Clase `Acelerometro`[3]

Esta clase implementa la interface nativa *SensorEventListener,* activa el sensor acelerómetro, uno de los métodos que contiene actualiza los datos de posición siempre que se produce un cambio de posición del dispositivo.

Para activar el sensor se necesita una instancia de la clase *SensorManager* y otra de la clase *Sensor*.

```
public Acelerometro(Context context) {
	sensorManager = (SensorManager)
	context.getSystemService(Context.SENSOR_SERVICE);

	if(sensorManager.getSensorList(Sensor.TYPE_ACCELEROMETER).size() !=0){
		accelerometer =
	sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
		sensorManager.registerListener(this, accelerometer,
	SensorManager.SENSOR_DELAY_GAME);
	}
}
```

Esta clase captura los eventos de movimiento del dispositivo y los procesa a través del siguiente método:

```
public void onSensorChanged(SensorEvent event) {
	if(accelerometer != null){
		x = event.values[0];
		y = event.values[1];
		z = event.values[2];
	}
}
```

4.3.2 Clase `Lienzo`

La clase *`Lienzo`* de este modo es similar al del modo automático, el único cambio realizado es la adición de un método para recoger los datos del acelerómetro.

```
public void velocidad(){
       ax=aceleracion.getAcelX();
       ay=aceleracion.getAcelY();
       vx=2*-ax;
       vy=2*ay;
}
```

Para el eje X se cambia el signo para un mejor funcionamiento.

Este método es llamado cada vez que se redibuja la pantalla, en el método *onDraw:*

```
public void onDraw(Canvas canvas) {
        if(canvas!=null){
             canvas.drawColor(Color.BLACK);
          velocidad();
             naves=listaNaves.getNaves();

             if(naves!=null){
                  velocidad();
                   for (int i = 0; i < naves.length; i++) {
                         if (naves[i] instanceof IDibujable){
                               ((IDibujable)naves[i]).dibujar(canvas);
                          }
                         if (naves[i] instanceof IObservadoresNaves){
                               IListaNaves lista = new ListaNaves();
                               for(int j = 0; j < naves.length; j++){
                                     if (j != i && distancia(naves[i],
naves[j]) < ((IObservadoresNaves)naves[i]).getAlcanceRadar()){
                                           lista.addNave(naves[j]);
                                     }
                               }
       ((IObservadoresNaves)naves[i]).setRadar(lista.getNaves());
                         }

                    }

             }
        }
    }
```

4.4 Visual XML

Para ambos modos la interface visual es la misma. Se presenta en la orientación del dispositivo, vertical y horizontal:

- La superficie de dibujo, la es toda la pantalla.
- Los controles de las diferentes opciones del juego.
- Los controles son simple botones situados en la parte superior de la visual.

A continuación se muestra la el contenido del fichero main.xml que define la visual:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent" >
<proyecto.naves_automatico.vistas.Lienzo
android:id="@+id/lienzo"
android:layout_width="wrap_content"
android:layout_height="wrap_content">
</proyecto.naves_automatico.vistas.Lienzo>
<GridLayout
android:layout_width="match_parent"
android:layout_height="match_parent"
android:columnCount="4" >
<Button
android:id="@+id/patrulla"
android:layout_column="0"
android:layout_gravity="left"
android:layout_row="0"
android:text="Patrullera" />
<Button
android:id="@+id/patera"
android:layout_column="1"
android:layout_gravity="left"
android:layout_row="0"
android:text="Patera" />
<Button
android:id="@+id/reset"
android:layout_column="3"
android:layout_gravity="left"
android:layout_row="0"
android:text="Reset" />
<Button
android:id="@+id/speedup"
```

```
android:layout_column="0"
android:layout_gravity="left"
android:layout_row="1"
android:text="Speed+" />
<Button
android:id="@+id/speeddown"
android:layout_column="1"
android:layout_gravity="left"
android:layout_row="1"
android:text="Speed-" />
</GridLayout>
</FrameLayout>
```

4.5 **Actividad principal[4]**

Una de la funciones de la clase principal de la actividad, Naves_automatico y Naves_manual es recuperar las instancias de los botones definidos por la visual XML y asignar al evento de pulsación de cada uno la lógica de

lanzamiento de la actividad asociada a dicho botón.

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);

        setContentView(R.layout.main);

        patrulla = (Button) findViewById(R.id.patrulla);
        patrulla.setOnClickListener(this);

        patera =(Button) findViewById(R.id.patera);
        patera.setOnClickListener(this);

        reset = (Button) findViewById(R.id.reset);
        reset.setOnClickListener(this);

        speedup = (Button) findViewById(R.id.speedup);
        speedup.setOnClickListener(this);

        speeddown = (Button) findViewById(R.id.speeddown);
        speeddown.setOnClickListener(this);

        lienzo= (Lienzo) findViewById(R.id.lienzo);
    }
```

5 Aplicación NAVES en Red

Este modo es una extensión del juego original con una pantalla de radar y posibilidad de ejecución simultánea de varios jugadores desde distintas plataformas móviles Android.

Esta variación requiere de un servidor Java al cual se conectaran distintos dispositivos Android.

Para el entendimiento de este modo empezare la explicación con el servidor Java.

5.1 Servidor aplicación Naves

Para la implementación del servidor se ha utilizado como base el juego original implementado en Java y se a añadido una funcionalidad de conexión a través de socket UDP, a continuación se explicara la forma de conexión de los dispositivos Android.

5.1.1 Clase `Conexion`

Esta clase implementa la interface *Runnable,* por consiguiente esta clase se ejecuta como un hilo aparte, en paralelo a la clase principal del servidor, de la cual se hablara mas adelante.

El constructor de esta clase requiere una instancia de la clase principal, para añadir una nueva nave cada vez que llegue una nueva conexión.

Para las conexiones se ha utilizado el protocolo de transporte UDP, por que se transmiten un gran volumen de datos y no es necesaria la confirmación de recibo.

Para abrir una conexión socket UDP se utilizan las clases `DatagramSocket`, `DatagramPacket` y `ByteBuffer`.

- DatagramSocket: representa un socket para enviar y recibir paquetes de datagramas.
- DatagramPacket: representa un paquete datagrama.
- ByteBuffer: almacena los datos en formato byte para enviar.

Contiene cuatro métodos más el constructor.

El constructor instancia el socket, dos DatagramPacket, uno para enviar y otro para recibir, y la clase principal.

```
public Conexion(Principal test){
	try{
		principal=test;
		socket = new DatagramSocket(PUERTO_DEL_SERVIDOR);
		dato = new DatagramPacket(lMsg, lMsg.length);
		envio = new DatagramPacket(lMsg,
lMsg.length,InetAddress.getByName("localhost"), PUERTO_DEL_CLIENTE);
	}catch (Exception e){
		e.printStackTrace();
	}
}
```

La implementación del método *run* de la interface *Runnable* es la siguiente:

```
public void run() {
	while (true){
		recibir();
	}
}
```

Simplemente llama al método *recibir* hasta el final de la aplicación.

El método *recibir* queda así:

```
public void recibir(){
	try{
		socket.receive(dato);
		System.out.print("Recibido dato de " +
dato.getAddress().getHostName() + " : \n");
		hosted=dato.getAddress().getHostName();
		guardar(dato.getData(), hosted);
	}catch (Exception e){
		e.printStackTrace();
	}
}
```

Como la función *receive* es bloqueante, por tanto este hilo se queda bloqueado hasta que recibe una nueva conexión, cuando esto ocurre se obtiene la dirección de donde viene, se recogen los datos y se guardan, para este cometido hay otro método que se llama *guardar.*

El método *guardar se* ocupa de procesar la información recibida, crear una nueva nave en la aplicación principal, crear un nuevo socket para recibir las actualizaciones de los datos e informar al dispositivo conectado el nuevo puerto de comunicación.

```
public void guardar(byte[] b, String h){
	try {
		buf = ByteBuffer.wrap(b);
		System.out.println("Guardando host");
			socketC=new DatagramSocket(PUERTO_DEL_CLIENTE+valor+1);
			System.out.println("nuevo socket: "+socketC);
			host=h;
			System.out.println("Guardando host: "+host);
			enviaPuerto(PUERTO_DEL_CLIENTE+valor+1, host);
			valor++;
			tipo=buf.getInt();
			id=buf.getInt();
			principal.addObjects(1, tipo,id, socketC, host);
			if(valor > 100 && principal.cantidadNaves() < 100){
				valor=0;
			}
	} catch (SocketException e) {
			// TODO Auto-generated catch block
		e.printStackTrace();
	}
    }
```

Y por ultimo el método *enviaPuerto* simplemente se encarga de informar al nuevo dispositivo del nuevo puerto para comunicarse con el servidor.

```
public void enviaPuerto(int puerto, String hst){
	try {
		bufenv= ByteBuffer.allocate(4);
		System.out.println("puerto "+puerto);
		bufenv.putInt(puerto);
		envpuerto=bufenv.array();

		envio.setData(envpuerto);
			envio.setLength(envpuerto.length);

			envio.setAddress(InetAddress.getByName(hst));
			socket.send(envio);

	} catch (UnknownHostException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
    }
```

5.1.2 Clase `Principal`

A continuación se explicará la clase principal del servidor, esta clase da interface visual en el PC y se encarga de la comunicación entre el servidor y los distintos dispositivos ya conectados.

Realiza la función de actualización de datos y llamada al refresco de pantalla mediante la clase *Timer* el cual se añade a una clase *TimeHandler* que implementa un ActionListener para actualizar los datos de las naves.

```
timer = new Timer((400 / (int) (Math.pow(2,rate))), this);
timer.addActionListener(new TimerHandler(timer));
timer.start();
```

Para obtener los datos de actualización de las naves se utiliza el método *recibir()* que comprueba el socket de cada nave registrada a la espera de recibir datos durante 50 milisegundos.

```
public void recibir(){
		try{
			lMsg = new byte[MAX_UDP_DATAGRAM_LEN];

			for(int i=0;i < objetos.length;i++){
				if(objetos[i] != null){
					try{

	((IObservadorNaves)objetos[i]).getSocket().setSoTimeout(50);

	((IObservadorNaves)objetos[i]).getSocket().receive(dato);
					datosVel(dato.getData(),i);
					((IObservadorNaves)objetos[i]).setborra(0);

					}catch(SocketTimeoutException er){

	((IObservadorNaves)objetos[i]).setborra(1);
					}
				}
			}
		}
		catch (Exception e){
			e.printStackTrace();
		}
		lMsg=null;
		return;
}
```

Si pasa 50 veces por un socket sin recibir nada la nave enlazada al socket será borrada.

```
if(((IObservadorNaves)objetos[i]).getborra() >= 50){

      if(borrado=listaNaves.deleteNave(objetos[i].getId())){
                  System.out.println("Borrada Nave: "+i);
      }
}
```

Una parte de la implementación de la función radar de la aplicación la aporta el método *enviarRadar*:

```
public void enviarRadar(int tipo, int idR, float dist, float angle, String h,
DatagramSocket s){
         try {
             bufenv= ByteBuffer.allocate(16);
             bufenv.putInt(idR);
             bufenv.putFloat(dist);
             bufenv.putFloat(angle);
             bufenv.putInt(tipo);
             cantNaves=bufenv.array();

             envio.setData(cantNaves);
             envio.setLength(cantNaves.length);

             envio.setAddress(InetAddress.getByName(h));
             s.send(envio);
             System.out.println("radar enviado a host: "+h);
             System.out.println("Distancia: "+dist+"\nAngulo: "+angle+"\nEn
grados: "+Math.toDegrees(angle));
        } catch (UnknownHostException e) {
             e.printStackTrace();
        } catch (IOException e) {
             e.printStackTrace();
        }
}
```

Cuando una nave entra dentro del radar de otra es detectado y se llama a esta función la cual envía a la nave la identificación, tipo y posición el coordenadas polares.

5.2 **Cliente aplicación Naves**

Similar al servidor, la base del cliente es la versión manual de la aplicación con el añadido de dotarle de conexión a través de un socket UDP.

Para implementar esta función se ha añadido una nueva clase, la clase *Conexión*, y se han modificado algunas clases, la modificación más significativa se ha realizado sobre la clase *Lienzo* y la clase *Nave_red*, esta clase se extiende de la clase *Activity,* es la clase principal.

El Diagrama de clases queda así:

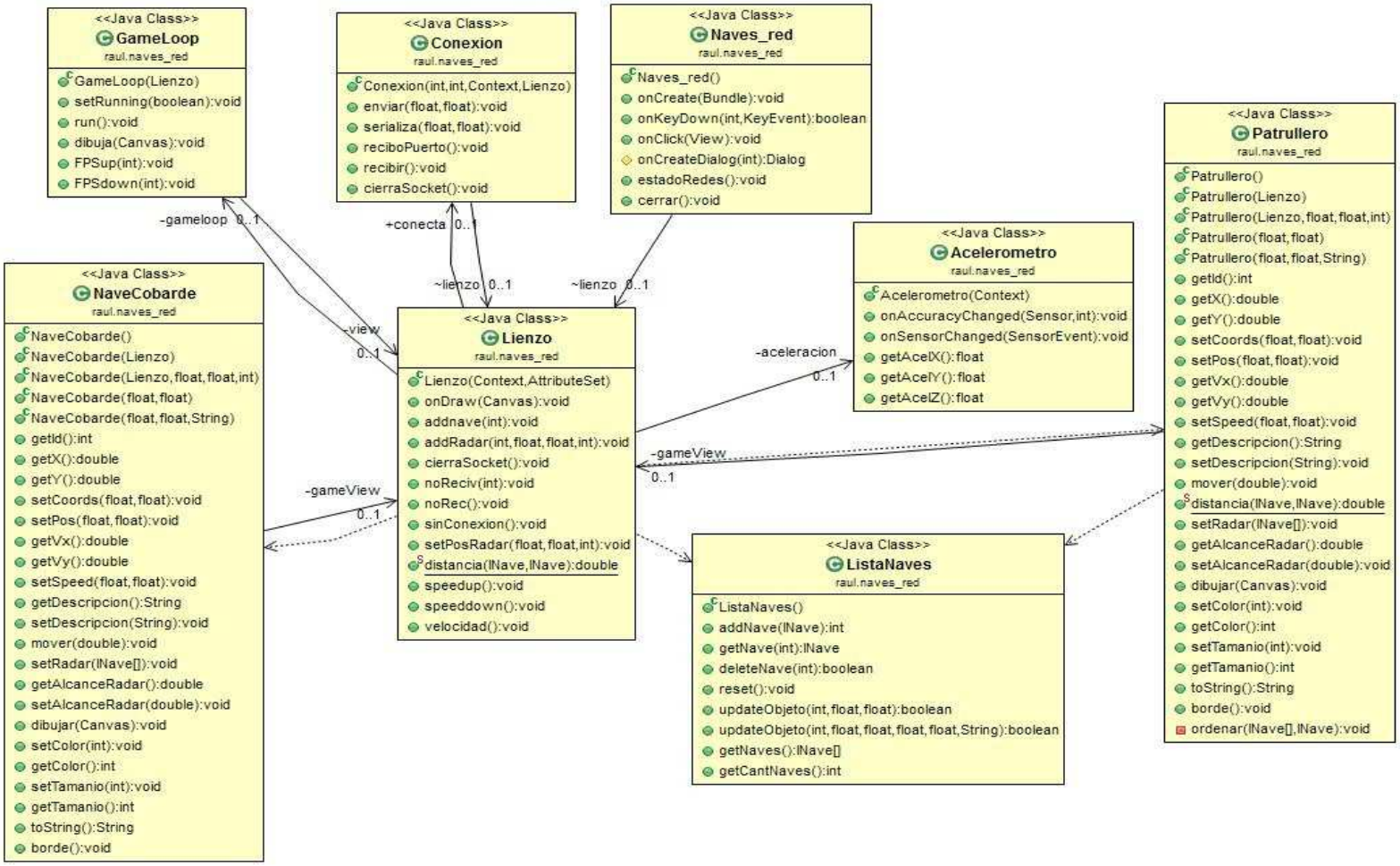

Figura 26. Diagrama de clases de la aplicación NAVES en red

5.2.1 Clase Conexion[5] [6]

Cuando se instancia esta clase, en el constructor se crea el socket y los paquetes UDP, una vez creados estos atributos, se envía la identificación y el tipo de nave al servidor, para que este responda con el nuevo puerto para una comunicación continua.

Los métodos mas relevantes de esta clase son *enviar(float vx, float vy)* y *recibir(),* el método *enviar* es llamado en cada redibujado de la clase *Lienzo* y envía al servidor los datos de aceleración de la nave, que serán los valores del acelerómetro, una vez enviados estos datos llama al método *recibir()*, este método espera 100milisegundos a la espera de recibir información del servidor, esta información es sobre las naves que hay dentro del radar, si no hay naves en el radar, no se recibirá nada.

```
public void enviar(float vx, float vy){
	try {
		if(puerto_serv != 5557){
			serializa(vx,vy);
			packenv.setData(arrayByte);
			packenv.setLength(arrayByte.length);
			sock.send(packenv);

			recibir();
			}

	} catch (SocketException e) {
		e.printStackTrace();
	}catch (UnknownHostException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

```
public void recibir(){
	i=0;
	try {
		sockenv.setSoTimeout(100);
		sockenv.receive(recibo2);
		lienzo.setcontrol("Recibiendo...");
		bufrec = ByteBuffer.wrap(recibo2.getData());

		id=bufrec.getInt(0);
		dis=bufrec.getFloat(4);
		angle=bufrec.getFloat(8);
		tipoR=bufrec.getInt(12);

		lienzo.setcontrol("ID ====> "+id+"\nTipo: "+tipoR);
		lienzo.posi(dis, angle);
```

```
            if(idR.size() == 0){
                  idR.add(id);
                  lienzo.addRadar(tipoR, dis, angle, id);
            }else{
                  for(i=0; i< idR.size(); i++){
                        if(id == idR.get(i)){
                              lienzo.setPosRadar(dis, angle, id);
                              break;
                        }
                  }
                  if(i >= idR.size()){
                        idR.add(id);
                        lienzo.addRadar(tipoR, dis, angle, id);
                  }

            }
            lienzo.noReciv(idR.size());
            bufrec=null;

      }catch(SocketTimeoutException e){
            lienzo.setcontrol("NO Recibo");
            lienzo.noRec();
            idR.clear();
      }catch (IOException e) {
                  // TODO Auto-generated catch block
            e.printStackTrace();
      }
}
```

5.2.2 Clase `Lienzo`

Las modificaciones de esta clase se han realizado para poder visualizar las naves de radar según recibe datos del servidor.Se ha añadido un método para añadir nuevas naves al radar de la nuestra y una nueva lista para almacenarlos.

```
public void addRadar(int tipo, float dist, float angulo, int id){
             if(tipo==1){
                    radar=new Patrullero(this, dist, angulo, id);
                    lista.addNave(radar);
                    contRad++;
             }
             if(tipo==2){
                    radar=new NaveCobarde(this, dist, angulo, id);
                    lista.addNave(radar);
                    contRad++;
             }
     }
```

5.2.3 Clase `Naves_red`

Esta clase se diferencia de las anteriores clases *Activity* es la eliminación de botones para añadir naves y reset del juego, y se a añadido una dialogo de selección para elegir el tipo de nave.

```
protected Dialog onCreateDialog(int id) {

      final String[] items = {"Patrulla", "Patera"};

        AlertDialog.Builder builder = new AlertDialog.Builder(this);

        builder.setTitle("Selecciona Nave");
        builder.setItems(items, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int item) {
             nave=item+1;
             switch(nave){
                      case PATRULLA:
                           lienzo.addnave(PATRULLA);
                           break;
                      case PATERA:
                           lienzo.addnave(PATERA);
                           break;
                      default:
                            //no hace nada
                    }
            }
        });

        return builder.create();

    }
```

5.2.4 Configuración y permisos

En el apartado dedicado al sistema operativo Android vimos que el acceso a las funcionalidades del sistema operativo se controla mediante permisos.

Para este juego no es necesario ningún permiso para las versiones automática e interactiva, pero para la versión del juego a través de red wifi es necesario dar permisos para que tenga acceso a la conexión wifi del dispositivo.

El código xml necesario para dar permiso, al juego, acceso a internet es el siguiente:

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

5.2.5 Visual XML

Se presenta en la orientación horizontal:

- La superficie de dibujo, la es toda la pantalla.
- Los controles de las diferentes opciones del juego.
- Los controles son simple botones situados en las esquinas superiores de la visual.

A continuación se muestra la el contenido del fichero main.xml que define la visual:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent" >
<raul.naves_red.Lienzo
android:id="@+id/lienzo"
android:layout_width="wrap_content"
android:layout_height="wrap_content">
</raul.naves_red.Lienzo>
<RelativeLayout
android:layout_width="match_parent"
android:layout_height="match_parent" >
<Button
android:id="@+id/speeddown"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignParentRight="true"
android:layout_alignParentTop="true"
android:text="Speed-" />
```

```
<Button
android:id="@+id/speedup"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignParentLeft="true"
android:layout_alignParentTop="true"
android:text="Speed+" />
</RelativeLayout>
</FrameLayout>
```

5.2.6 Actividad principal

Una de la funciones de la clase principal de la actividad, Naves_red, es recuperar las instancias de los botones definidos por la visual XML y asignar al evento de pulsación de cada uno la lógica de lanzamiento de la actividad asociada a dicho botón.

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.main);

        speedup = (Button) findViewById(R.id.speedup);
        speedup.setOnClickListener(this);

        speeddown = (Button) findViewById(R.id.speeddown);
        speeddown.setOnClickListener(this);

        lienzo= (Lienzo) findViewById(R.id.lienzo);
        showDialog(DIALOGO_SELECCION);
        estadoRedes();
    }
```

Y también se comprueba si esta la conexión wifi activa:

```
public void estadoRedes() {
        ConnectivityManager connMgr = (ConnectivityManager)
this.getSystemService(Context.CONNECTIVITY_SERVICE);
        final android.net.NetworkInfo wifi =
connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);

        if (!wifi.isAvailable()){
             Toast.makeText(this, "Wifi inactivo, activelo",
Toast.LENGTH_LONG).show();
             finish();
        }
    }
```

En este modo se ha añadido una función adicional, de cerrar los sockets, al pulsar el botón de ATRÁS del dispositivo:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event)
      {
            if ((keyCode == KeyEvent.KEYCODE_BACK))
            {
                  cerrar();

            return true;
            }
      return super.onKeyDown(keyCode, event);
     }
```

5.2.7 ¿Qué ocurre si no hay conexión?

Se pueden dar dos casos en los que no consiga el cliente conectar con el servidor.

- **Cliente no tiene la señal wifi activada**, si ocurre esto se muestra un mensaje temporal informando de tal estado y el programa finaliza.
- **El servidor no esta activo**, en este caso simplemente se muestra un mensaje temporal, y no de llega a dibujar nada en pantalla, solo un mensaje que informa del estado.

6 Fortalezas, carencias y mejoras

A lo largo de esta memoria se ha explicado el proceso de desarrollo de la aplicación NAVES para dispositivos móviles bajo el sistema operativo Android. En este capitulo se va a analizar cuales son los puntos fuertes, las características del programa que se considera que dan valor de cara al usuario. También se dará una revisión a los aspectos de la aplicación que pueden ser mejorables, ya sea a nivel teórico, técnico o de usabilidad, y posibles maneras de perfeccionarlos.

6.1 Punto fuerte: Compatibilidad multidispositivo

Un aspecto remarcable de nuestra aplicación es que, por el simple motivo de haber elegido Android como plataforma objetivo, la aplicación NAVES puede ejecutarse sin cambios en cualquier dispositivo que utilice el sistema operativo Android, lo cual nos abre un gran abanico de posibilidades. Actualmente la cantidad de dispositivos Android es muy variada y ocupa la mayor parte de dispositivos que se venden al año, cuando se habla de dispositivos, se esta hablando tanto de *Smartphone* como de *tablets*, por que actualmente el mercado de los tablets se esta asentando en la sociedad con gran aceptación debido a la calidad de estas y su bajo precio en la parte de estas. Si en vez de desarrollar una aplicación para Android la hubiéramos desarrollado para iOS, por ejemplo, nuestro público objetivo quedaría reducido solamente a aquellos usuarios que posean un dispositivo iPod, iPhone o iPad; dado que estos modelos, fabricados exclusivamente por Apple, suelen situarse en el rango superior de precios dentro de sus respectivos segmentos, se puede esperar que el nuevo se usuarios potenciales se vea reducido.

6.2 Punto fuerte: Facilidad de manejo

El manejo de este juego es muy simple, con solo mover el dispositivo nuestras naves se moverán, gracias al sensor de aceleración de estos, se usa esta función para dejar una pantalla más vistosa y limpia, a diferencia del uso de botones que utilizan espacio en la pantalla y una lógica mas compleja.

6.3 Punto fuerte: Implementación simple

Dado que este proyecto esta orientado a la enseñanza del framework de Android, este juego es sencillo para el rápido aprendizaje de los alumnos a dibujar formas geométricas, utilización de sensores y conexión por sockets.

6.4 A mejorar: Dificultad de implementar un patrón MVC puro

Antes de comenzar con el desarrollo de la aplicación, nos fijamos el objetivo de utilizar en el diseño de la misma el patrón Modelo-Vista-Controlador, a fin de poder tomarlo como ejemplo didáctico de implementación de un patrón de diseño conocido en un programa real. Pero, al tratar de aplicar el patrón de manera efectiva durante el desarrollo, hemos visto que el diseño de la arquitectura de Android no facilita esta tarea, debido sobre todo al alto acoplamiento existente entre el Controlador y la Vista en algunas de las clases básicas. Al no poder implementar correctamente el patrón MVC, el objetivo de utilizar nuestro código como material didáctico representativo no se ve satisfecho en su totalidad.

6.5 Mejora: Uso de patrones derivados

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

6.6 A mejorar: Poco vistoso

Debido al corto periodo de tiempo para la realización de este proyecto se ha implementado con figuras geométricas y una interface visual muy simple. Pero se justifica con la simplicidad obtenida.

6.7 Mejora: Utilización de Sprites

Los sprites son imágenes en 2D que se coloca sobre una imagen de fondo, se utilizan, normalmente, en los videojuegos y pueden representar AI (Artificial Intelligent) Entidades o el carácter principal controlado por el jugador.

Estos sprites se utilizarían en sustitución de las figuras geométricas que representas las naves.

6.8 **A mejorar: Falta de opciones de configuración**

Para apoyar la simplicidad de la aplicación se han suprimido estas opciones, pero a pesar de ello a salido una aplicación bastante completa para su objetivo final, la utilización didáctica.

6.9 **Mejora: Añadir opciones de configuración**

Se podrían añadir opciones como pausa, elección del color de la nave, posición de salida, etc. Aunque esto implicaría una complejidad de la lógica añadida por cada opción.

7 Posibles ampliaciones

Para mejorar su utilización y abarcar un mayor campo de la enseñanza se podrían hacer las siguientes ampliaciones:

7.1 Utilización del modo en red a través de internet

En este proyecto el modo en red se limita a una red wifi, esta ampliación consiste en cambiar la lógica de la aplicación para utilizar un servidor publico y eliminar esta limitación, con esta ampliación se consigue una mayor numero de jugadores y gracias a las redes de internet móvil, no es necesario estar en el rango de alcance del router servidor.

7.2 Mejorar la algoritmia lógica del modo automático

La lógica de IA del modo automático es simple, se puede realizar un algoritmo más complejo para evitar obstáculos o realizar una mejor huida.

7.3 Sistema de puntuación

Para crear un ambiente de competitividad se puede añadir un sistema de puntuación (score).

8 Conclusiones

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

8.1 Cumplimiento de objetivos

Se ha adaptado el juego original a Android y se ha documentado de forma que pueda servir para preparar docencia.

Se han implementado los tres modos de funcionamiento previstos: automático, manual, juego en red. Cada uno de los modos de funcionamiento ha servido además para explicar un conjunto de características de la aplicación: actualización de los datos y refresco de pantalla, vistas XML, captura de eventos en la interface visual, uso de sensores, como el acelerómetro y utilización de sockets, en concreto socket UDP, en Android.

8.2 Facilidad de desarrollo en Android

Desarrollar aplicaciones para Android es fácil. Sin lugar a dudas, esa sería la primera conclusión que nos viene a la cabeza tras evaluar todo el proceso de formación previa, obtención de las herramientas necesarias y programación en sí misma. En lo que a formación respecta, el equipo de Android en Google ha hecho un excelente trabajo de documentación de su plataforma, con artículos y ejemplos que abarcan todos los puntos de vista: desde el general del usuario que sólo quiere conocer las posibilidades de su dispositivo, hasta los detalles técnicos sobre gestión de memoria que necesita un programador para optimizar el rendimiento de su aplicación. Además, la utilización de tecnologías y estándares abiertos ha propiciado el surgimiento de una cantidad impresionante de sitios web, blogs, foros e incluso canales de YouTube y podcasts dedicados a la programación en Android; aficionados, desarrolladores independientes y empresas por igual comparten técnicas y conocimiento que facilitan enormemente los primeros pasos y el aprendizaje de cualquiera que esté interesado. Las herramientas software necesarias, ya hemos visto anteriormente que están disponibles a un coste cero: kit de desarrollo y librerías Android, librerías Java, editor Eclipse integrado con el plugin ADT... Todas ellas herramientas profesionales con características muy avanzadas, compatibles con casi cualquier ordenador personal (cosa que no sucede con otras plataformas) y que podemos obtener con unos simples clics de manera gratuita. Pero ¿y

las herramientas hardware? Desarrollar aplicaciones usando simuladores está muy bien, pero siempre es recomendable disponer físicamente de un terminal en el que poder probar los programas. Afortunadamente, la variedad de fabricantes y operadores que han dado su apoyo al sistema Android hace que hoy día sea realmente fácil conseguir un dispositivo, ya sea un teléfono móvil o un dispositivo de tipo tablet, por un precio muy reducido en comparación con los precios de terminales de otros sistemas, como Blackberry o iPhone. La facilidad del proceso de codificación en sí mismo es una característica algo más subjetiva, puesto que depende de los conocimientos y la destreza de cada programador. Sin embargo, aquí Java parte con ventaja: no solo es uno de los lenguajes más utilizados tanto en entornos académicos como productivos, sino que de hecho, cuando apareció Android, Java llevaba ya casi 10 años siendo utilizado por los desarrolladores de aplicaciones móviles para Symbian; con lo que es muy probable que cualquier programador que se inicie en Android tenga ya experiencia previa en Java. Otros factores, como la modularidad que proporcionan las Actividades o la sencillez de creación de las interfaces gráficas mediante XML, contribuyen también a esta facilidad de codificación. Todo lo anterior unido da como resultado que un programador medio, sin experiencia previa en desarrollo de aplicaciones móviles, pueda montar todo el entorno necesario, diseñar y codificar su primera aplicación Android, con funcionalidad real, en un fin de semana. Y algo que no hemos comentado hasta ahora: todo este proceso puede llevarse a cabo libremente. Google no requiere que los desarrolladores de aplicaciones Android se registren como tales, salvo que quieran publicar sus aplicaciones en Google Play; así, cualquiera puede obtener acceso a todas las herramientas y documentación necesarias, hacer sus primeras aplicaciones de prueba o incluso diseñar una aplicación para un proyecto sin necesidad de darle sus datos a ninguna empresa ni pagar ningún tipo de cuota.

8.3 **Fácil implementación de juegos 2D**

La implementación de un juego simple el 2D es relativamente sencilla, como se ha comprobado durante la explicación de este proyecto, con pocas clases pero bien definidas se puede realizar un juego simple en un tiempo considerable, esto abre una puerta a la oferta de juegos de esta plataforma, aunque también crea una gran competencia entre desarrollares, algo que en mi opinión es bueno, por cuanta mas competencia más esfuerzo por parte de los desarrolladores y por tanto mejores juegos aparecerán en el mercado.

En resumen, a lo largo de este proyecto se a realizado un juego para la plataforma Android didáctico para el desarrollando y entretenido para el usuario, con varios modos de juego. Y gracias a la gran similitud Android y Java la adaptación del juego original en Java a la plataforma Android a sido relativamente rápida y sencilla, a partir de ahí las distintas ampliaciones se han basado en esta adaptación.

9 Bibliografía

9.1 Utilizada durante la fase de desarrollo

[2]Android Game Programming tutorials [en línea].
http://www.edu4java.com/androidgame.html -> Consultado en mayo de 2012

[3]Desarrollo Avanzado de Aplicaciones para Dispositivos Móviles Android, Tema 6 [en línea].
http://ants.dif.um.es/~felixgm/docencia/android-> Consultado junio 2012

[4]Android Developer: The Developer's Guide [en línea].
http://developer.android.com/guide/components/index.html ->Consultado Julio 2012

[5]Android Developer: DatagramPacket class [en línea].
http://developer.android.com/reference/java/net/DatagramPacket.html ->Consultado agosto 2012

[6]Android Developer: DatagramSocket class [en línea].
http://developer.android.com/reference/java/net/DatagramSocket.html ->Consultado agosto 2012

9.2 Sobre aplicaciones móviles

GSM Arena: Samsung I9100 Galaxy S II [en línea]. Disponible en http://www.gsmarena.com/samsung_i9100_galaxy_s_ii-3621.php ->Consultado en agosto de 2012.

PowerBook G4 [en línea. Diponible en http://apple-history.com/pg4 -> Consultado en agosto de 2012.

Wikipedia: Mobile operating system [en línea]. Disponible en http://en.wikipedia.org/wiki/Mobile_operating_system -> Consultado en septiembre de 2012.

Wikipedia: History of mobile phones [en línea]. Disponible en http://en.wikipedia.org/wiki/History_of_mobile_phones ->Consultado en septiembre de 2012.

Wikipedia: PDA [en línea]. Disponible en http://es.wikipedia.org/wiki/PDA -> Consultado en septiembre de 2012.

Wikipedia: Wireless Application Protocol [en línea]. Disponible en http://en.wikipedia.org/wiki/Wireless_Application_Protocol -> Consultado en septiembre de 2012.

Ciberaula: Introducción a J2ME [en línea]. Disponible en http://www.ciberaula.com/curso/javabasico ->. Consultado en septiembre de 2012.

Wikipedia: Smartphone [en línea]. Disponible en http://en.wikipedia.org/wiki/Smartphone ->Consultado en septiembre de 2012.

Wikipedia: Symbian OS [en línea]. Disponible en http://en.wikipedia.org/wiki/Symbian_OS ->Consultado en septiembre de 2012.

Wikipedia: BlackBerry [en línea]. Disponible en http://en.wikipedia.org/wiki/BlackBerry -> Consultado en septiembre de 2012.

[1]Comparativa de sistemas operativos [en linea]. Disponible en http://mobithinking.com/blog/2011-handset-and-smartphone-sales-big-picture ->Consultado en septiembre de 2012

9.3 **Sobre Android**

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

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

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

9.4 **Sobre patrones de diseño**

Reenskaug, Trygve: MVC [en línea]. Disponible en http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html ->consultado junio 2012

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

9.5 **Otros**

[8]PFC Aplicación para dispositivos móviles Android: Guía de los edificios de la Universidad Politécnica de Cartagena, por Luis Gonzalo Soto Aboal