

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA.



Proyecto Fin de Carrera

Prototipos de prueba de acelerómetro y conexión
Bluetooth para terminales Android.

Autor: Raúl Velasco Gracia

Director: Juan Ángel Pastor Franco

Co-directora: María Francisca Rosique Contreras

A Yolanda, mi amor, por ser siempre
un apoyo en este largo camino.
A mi familia y todos aquellos que me
ayudaron a llegar hasta aquí.

Índice de Contenidos.

| | |
|--|--------|
| Introducción | - 9 - |
| Motivación | - 9 - |
| Objetivos | - 10 - |
| Requisitos..... | - 10 - |
| CAPITULO II. ANDROID | - 11 - |
| 1.1 Introducción. | - 11 - |
| 1.2 ¿Qué es Android?..... | - 11 - |
| 1.3 La plataforma de desarrollo: Eclipse..... | - 11 - |
| 1.4 El SDK de Android. | - 13 - |
| 1.5 Plugin para Eclipse..... | - 17 - |
| 1.6 Ejecución y depuración. | - 20 - |
| 1.6.1 Preparación de un terminal Android para la depuración y ejecución de aplicaciones..... | - 20 - |
| 1.6.2 Emulación de un terminal Android para la depuración y ejecución de aplicaciones..... | - 21 - |
| Capítulo II. Creación de aplicaciones Android. | - 25 - |
| 2.1 1ªAplicación. Números primos. | - 25 - |
| 2.1.1 Elementos de un proyecto Android..... | - 30 - |
| 2.1.2 Modificar el layout de una Activity. | - 32 - |
| 2.1.3 La clase Activity..... | - 34 - |
| 2.1.3.1 El ciclo de vida de activity. | - 34 - |
| 2.1.3.2 Estados y métodos de activity. | - 34 - |
| 2.1.4 Probar una aplicación desde Eclipse. | - 40 - |
| 2.1.5 Depuración de aplicaciones con Eclipse | - 43 - |
| 2.1.6 Mejora de aplicaciones..... | - 47 - |
| 2.1.6.1 Gestión del evento de pulsar un botón. | - 47 - |
| 2.1.6.2 Generar y gestionar eventos con el teclado virtual..... | - 48 - |
| 2.1.7 La clase View y sus Subclases. | - 50 - |
| 2.2 2ªAplicación. Servicio..... | - 51 - |
| 2.2.1 Toast..... | - 52 - |
| 2.2.2 La clase Service..... | - 52 - |
| 2.2.2.1 El ciclo de vida de Service. | - 52 - |
| 2.2.2.2 Elementos del Service de la aplicación Servicios. | - 54 - |
| 2.2.3 Messenger, Handler, HandlerMessage y Message..... | - 56 - |

| | |
|---|---------|
| 2.2.4 Intent | - 58 - |
| 2.2.4.1 Estructura de un intent..... | - 58 - |
| 2.2.5 Funcionamiento de la Aplicación Servicio. | - 59 - |
| 2.3 3ª Aplicación. Sensores y Bluetooth | - 66 - |
| 2.3.1 Crear un proyecto a través de un ejemplo. | - 67 - |
| 2.3.2 Funcionamiento del AccelerometerPlay | - 68 - |
| 2.3.3 Sensores..... | - 77 - |
| 2.4 AcBlueServer y AcBlueClient. | - 78 - |
| 2.4.1 Menú de opciones..... | - 78 - |
| 2.4.2 onCreateOptionsMenu. | - 80 - |
| 2.4.3 Creación de una segunda activity. | - 80 - |
| 2.4.3.1 Creación de layout de la segunda activity. | - 81 - |
| 2.4.3.2 Clase Ventana de Control..... | - 82 - |
| 2.4.4 Intent Filter y BroadcastReceiver. | - 82 - |
| 2.4.5 Clase onItemClickListener | - 84 - |
| 2.4.6 Método onCreate y onDestroy | - 84 - |
| 2.4.7 Modificar el AndroidManifest.xml | - 85 - |
| 2.4.8 onOptionsItemSelected. | - 86 - |
| 2.4.9 startActivity, startActivityForResult y onActivityResult. | - 87 - |
| 2.5 ServidorBluetooth | - 89 - |
| 2.5.1 Constructor ServidorBluetooth. | - 90 - |
| 2.5.2 Métodos de ServidorBluetooth..... | - 91 - |
| 2.5.3 AcceptThread. | - 94 - |
| 2.5.4 ConnectThread. | - 95 - |
| 2.5.5 ConnetedThread. | - 96 - |
| 2.5.6 Modificaciones comunes en cliente y servidor de la activity AccelerometerPlay. | - 98 - |
| 2.5.7 Modificaciones en el servidor de la activity AccelerometerPlay. | - 99 - |
| 2.5.8 Modificaciones en el cliente de la activity AccelerometerPlay..... | - 100 - |
| 2.6 4ª Aplicación. Esqueleto de aplicación para teleoperación. | - 101 - |
| 2.6.1 Creación de una activity de paso. Inicio..... | - 101 - |
| 2.6.2 Activity Menú. | - 102 - |
| 2.6.3 Ventanas de Dialogo. | - 103 - |
| 2.6.4 Gestión del botón físico (o virtual) Back. | - 104 - |
| 2.6.5 Ciclo de vida de la Activity Menú..... | - 105 - |

| | |
|---|----------------|
| 2.6.6 Activity Conectar. | - 106 - |
| 2.6.8 Sensor orientation Vs accelerometer y magnetic field. | - 108 - |
| CAPITULO III. ESTUDIO DEL SIMULADOR DEL ROBOT PIONEER. MOBILESIM | - 111 - |
| 3.1 Instalación del simulador. | - 111 - |
| 3.2 Iniciar una simulación. | - 112 - |
| 3.3 Inconvenientes del simulador. | - 113 - |
| Conclusiones | - 114 - |
| Facilidad de desarrollo en Android | - 114 - |
| El funcionamiento del Bluetooth..... | - 115 - |
| Trabajos futuros. | - 115 - |
| Bibliografía. | - 116 - |

Introducción

Motivación

A lo largo de los estudios de Ingeniería Técnica de Telecomunicación, Especialidad en Telemática he estudiado diferentes lenguajes de programación y mecanismo de comunicación, además de ser usuario de dispositivos móviles con sistema operativo Android. Todo esto me provocó la inquietud de saber desarrollar aplicaciones para dicho sistema. Cuando empecé a investigar el entorno de desarrollo, descubrí la gran cantidad de información que había pero de manera muy dispersa o excesiva, por eso decidí que este proyecto final de carrera no solo fuera simplemente el desarrollo de una aplicación para Android sino un manual para todo aquel que quisiera iniciarse en este maravilloso mundo.

Los dispositivos móviles actuales son la versión moderna de la navaja suiza, dispone de diferentes herramientas de comunicación como Bluetooth, 3G, Wifi, NFC... herramientas de medición como Acelerómetro, Giroscopio, Barómetro... y todo esto y mucho más concentrado en un dispositivo que cabe en la palma de la mano, tal es el abanico de posibilidades que en algunas misiones espaciales han sido utilizados como un aparato más de medición, como por ejemplo se puede ver en la siguiente imagen.



Fig. 1. Imagen de un terminal Android en el espacio.

Tras todo esto decidí la creación de diferentes aplicaciones usando las herramientas que se ofrecían como se podrá ver a lo largo de esta memoria además de iniciar un proceso de investigación en la teleoperación de robots haciendo uso de las herramientas de los dispositivos.

Objetivos

- Estudio de la plataforma Android y el empleo del acelerómetro y el empleo del acelerómetro y opcionalmente el giroscopio para enviar ordenes a un robot móvil.
- Realización y prueba de una aplicación que haga uso de tales recursos.
- Elaboración de una memoria que pueda utilizarse con propósito docente.

Requisitos

Para poder seguir el desarrollo de la memoria se da por supuesto que todo lector de esta memoria tiene unos conocimientos mínimos de lenguajes de programación de alto nivel, con orientación a objetos como Java (ó similares, como C++) y lenguajes de marcas como XML, además se recomienda disponer de un dispositivo con sistema operativo Android 3.0 o superior.

CAPITULO II. ANDROID

1.1 Introducción.

En estos capítulos se presentará al sistema operativo Android desde un punto didáctico, desde la instalación del entorno de desarrollo para la creación de nuevas aplicaciones, pasando por pequeños ejemplos funcionales y terminando por una aplicación capaz de intercambiar información de sensores a través del Bluetooth.

1.2 ¿Qué es Android?

Es un sistema operativo con un núcleo basado en Linux enfocado para ser utilizado en dispositivos móviles, aunque se ha empezado a introducir en diferentes dispositivos como por ejemplo televisores. Tras Android hay una alianza de 84 compañías para su desarrollo y mantenimiento llamada Open Handset Alliance liderada por Google y que se dedica al desarrollo de estándares abiertos para dispositivos móviles.

Aunque el desarrollo de aplicaciones se podría realizar sobre el propio núcleo en lenguaje C o C++, Android nos ofrece a Dalvik, su propia maquina virtual optimizada para requerir poca memoria y diseñada para ejecutar varias instancias de la maquina virtual, delegando al núcleo subyacente el soporte de aislamiento entre procesos.

Dalvik es como una maquina virtual java, ya que no opera directamente con los .class de Java compilados sino que estos son transformados en el formato de archivos Dex para ello nos ofrece la herramienta dx, que se encarga de realizar esta tarea de manera transparente al desarrollador.

La clave del éxito de Android es haber creado un sistema multiplataforma, libre y gratuito, en el que para programar en este sistema operativo ó incluirlo en un dispositivo no hay que pagar nada, lo cual ha permitido la creación de una gran comunidad de desarrolladores en la que no solo se crean nuevas aplicaciones para él sino que también lo han mejorado mediante nuevas versiones del Kernel o del mismo sistema operativo.

1.3 La plataforma de desarrollo: Eclipse

La ventaja de disponer de un sistema operativo abierto, es disponer de una de las herramientas más potentes para el desarrollo de software de forma gratuita. A lo largo de esta apartado explicaremos como instalarlo y configurarlo para poder crear nuestros propios proyectos Android en sistema operativo Windows 7 y Linux (utilizaremos 12.04).

Lo primero será descárgalo la última versión (a fecha de 06/08/12) es Eclipse Juno 4.2.

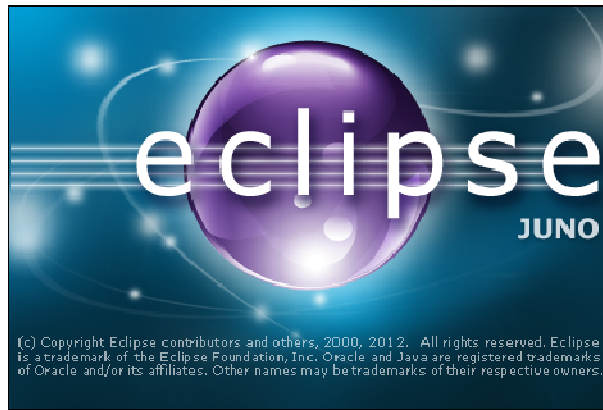


Fig. 2. Logo de Eclipse Juno

En la página <http://www.eclipse.org/downloads/> se podrá encontrar, se tendrá que indicar para que sistema operativo y para que arquitectura, en este PFC utilizaremos Eclipse IDE for JAVA EE Developers, aunque cualquiera de las otras versiones es válida.



Fig. 3. Página principal para la descarga de cualquier versión de Eclipse.

Además si se trabaja en Ubuntu podemos descargarlo a través del centro de software.

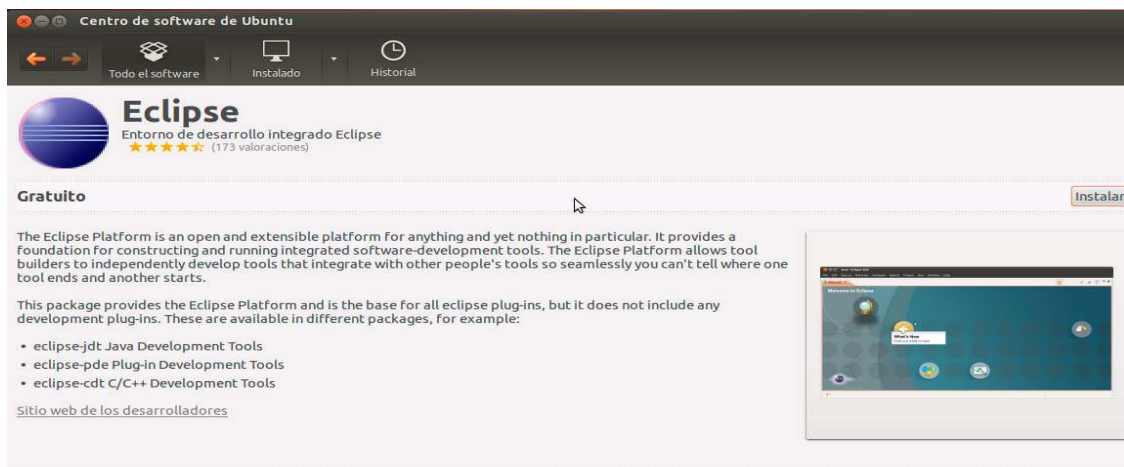


Fig. 4. Instalación de Eclipse en Ubuntu

Una vez descargado lo único que se tendrá que hacer es descomprimir el paquete y ejecutar.

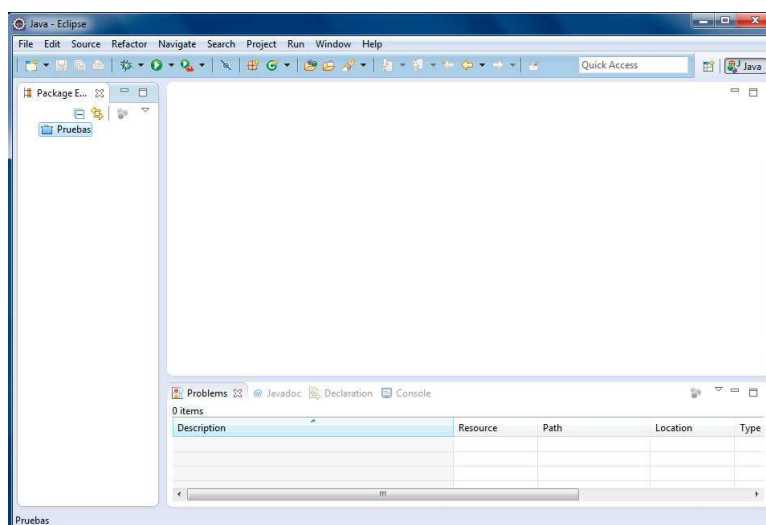


Fig. 5. Entorno de desarrollo eclipse sin ningún plugin

1.4 El SDK de Android.

Una vez que disponemos del entorno de desarrollo, lo siguiente será descargar e instalar el Kit de desarrollo de Software de Android (del inglés software development kit, SDK) en nuestro equipo.

Para ello Google pone a disposición la página <http://developer.android.com> donde explicaremos como descargarlo.

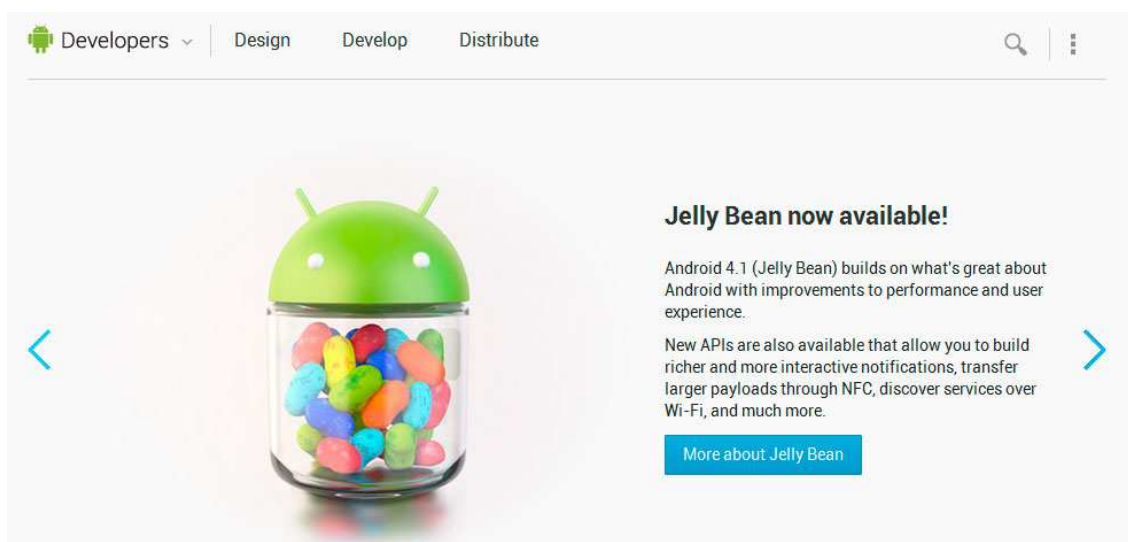


Fig. 6. Página principal de <http://developer.android.com>

Como se puede observar hay una pestaña de nombre Develop para encontrar el enlace para descarga del SDK se debe pinchar en dicho enlace en la que se desplegara una barra de pestañas nuevas con varias opciones y se vuelve a pinchar en la que ponga Tools, una vez realizado esto se mostrará la siguiente página.

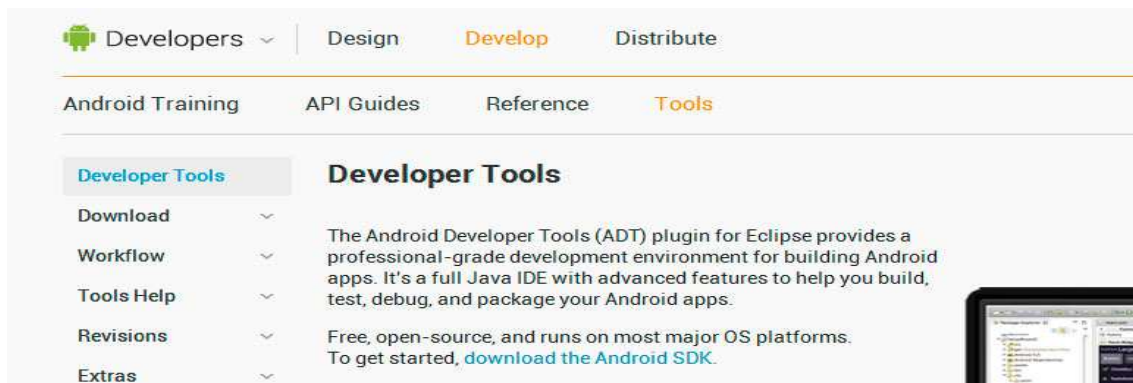


Fig. 7. Página de herramientas para desarrolladores de Android

Si se lee el primer párrafo se observa que se indica la existencia de "The Android Developer Tools plugin for Eclipse" el cual se explicará cómo se instala en nuestro Eclipse este plugin para poder utilizar este IDE (Entorno de Desarrollo Integrado). Dicho esto lo que se tendrá que hacer será utilizar el enlace "download the Android SDK", aunque también se podrá encontrar navegando por la paleta izquierda. Este enlace nos conducirá a la siguiente página.

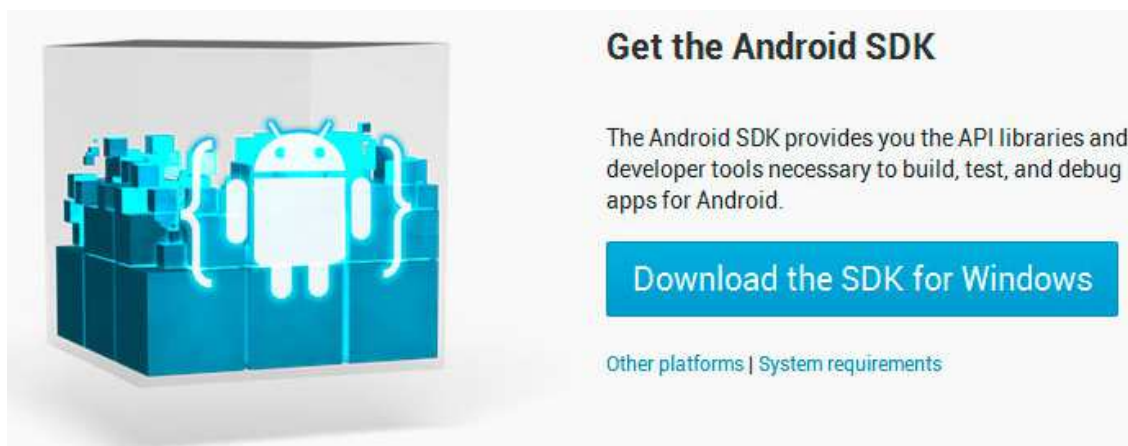


Fig. 8. Página para la descarga del instalador del SDK de Android para Windows.

En dicha página se podrá descargar el instalador del SDK de Android para el sistema operativo que se elija, en la imagen se puede ver el caso de Windows, no hay mayor diferencia para cualquier otro, una vez se pincha en el enlace se abrirá un dialogo de descarga.

Antes de iniciar la instalación del SDK de Android es recomendable tener instalado la maquina virtual de Java en el sistema operativo, en caso de no tenerla instalada se puede descargar de <http://www.java.com/es/download/>, ya que en caso de que esta no estuviera instalada la instalación del SDK no se podría completar.

Una vez que se ha asegurado la existencia de la maquina virtual Java en el sistema pasamos a la instalación del SDK, para ello solo se tendrá que ejecutar el archivo descargado y seguir los pasos que se nos indique en la ventana de instalación.



Fig. 9. Instalador de las herramientas del SDK de Android.

En Ubuntu no se tendrá que instalar el SDK Tools simplemente se descomprime en la ubicación que se desea y después desde un terminal se abrirá la carpeta tools y ejecutaremos como administrador el ejecutable Android mediante "sudo ./Android" (Nota. Mirar que se disponen de permiso de ejecución).

Terminada la instalación se podrá ver que se han instalado dos aplicaciones en Windows



Fig. 10. Android SDK Tools

La primera de ellas un gestor de maquinas virtuales Android que se verá más adelante y el segundo, el que nos interesa, el SDK Manager, que tras los pasos realizados hasta ahora todavía no se tiene instalado ningún SDK de Android, lo que hemos hecho por ahora ha sido instalar el gestor para poder instalar SDK's correspondientes a cada versión de Android en que queramos desarrollar nuestra aplicación.

Recordemos que Android va actualmente por su cuarta versión y que dentro de cada versión existe diferentes versiones, además algunos fabricantes personalizan algunas de estas añadiendo nuevas interfaces de programación de aplicaciones (API del ingles Application Programming Interface) por eso se nos ofrece el SDK Manager, el cual podemos ver en la siguiente imagen.

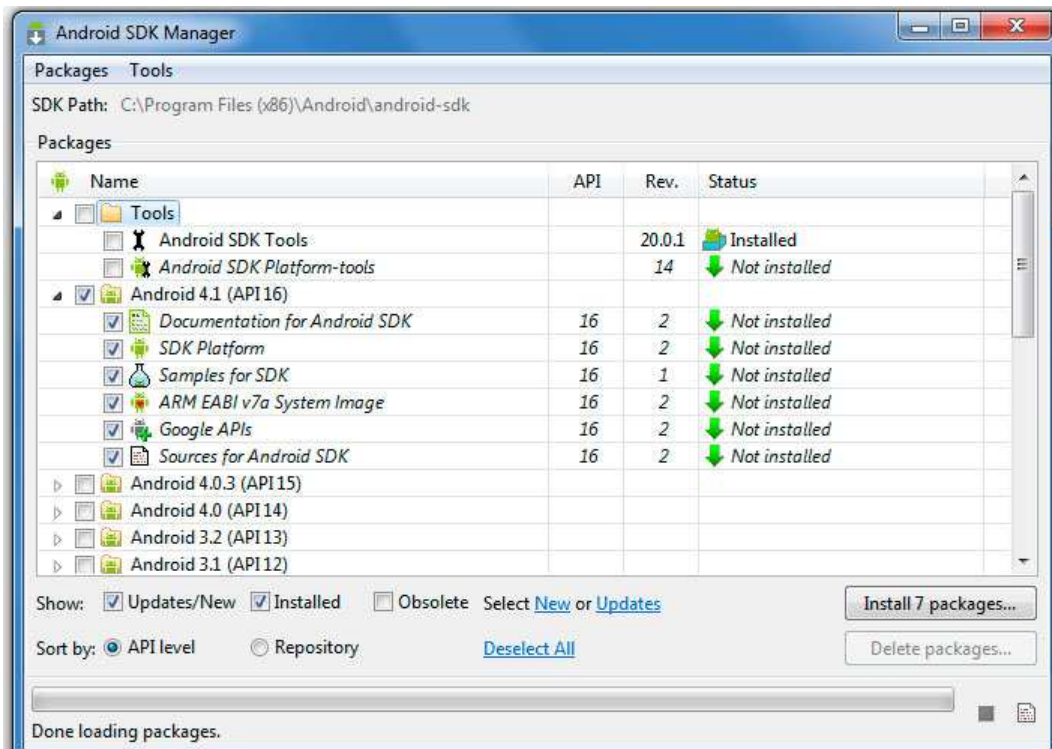


Fig. 11. Android SDK Manager.

Como podemos ver en la imagen superior, deberemos de marcar lo que deseamos instalar de cada versión del sistema operativo, ya que si quisiéramos podríamos instalar únicamente el "SDK Platform" de cada API, aunque lo recomendable es instalar todo el contenido.

Si no queremos bajar todas las versiones, la versión más recomendable es la 2.1 (API 7) ya que a día de hoy apenas quedan equipos con versiones inferiores a esta y además es compatible con las versiones posteriores, aunque si por ejemplo vamos a desarrollar aplicaciones para tablets Android lo mejor sería utilizar la versión 3.0 (API 11) ya que la gran mayoría de tablets del mercado utilizan esta o versiones superiores.

Por norma general cuando se desarrolla un proyecto para difundirlo en la tienda de aplicaciones de Google es común ofrecerlo desarrollado en diferentes versiones para que se adapte al sistema operativo del cliente.

Una vez tomada la decisión de que paquetes vamos a instalar se nos mostrara el siguiente dialogo, simplemente tendremos que aceptar las licencias, añadir que el tiempo de descarga variara dependiendo cuantos paquetes habremos marcados y la velocidad de nuestra conexión siendo el tiempo estimado de descarga de todos los paquetes de aproximadamente una hora.

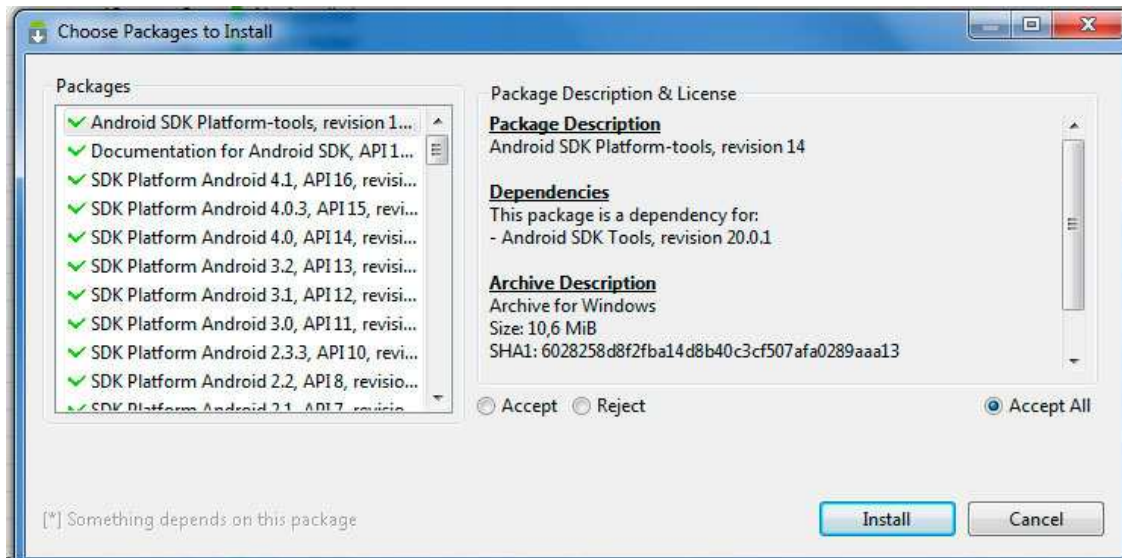


Fig. 12. Confirmación de instalación de paquetes y aceptación de licencias

1.5 Plugin para Eclipse

Como ya comentamos tras la Fig.6 y como literalmente nos indica Google "El Android Developer Tools (ADT) plugin para Eclipse ofrece un entorno de desarrollo a nivel profesional para la creación de aplicaciones Android. Se trata de un completo IDE para Java con características avanzadas para ayudarle a construir, probar y depurar el paquete de sus aplicaciones Android".

Lo que nos está indicando es que se nos ofrece una herramienta capaz de ayudarnos a nuestro trabajo como programador, mediante sugerencias, señalización y corrección de errores, generación de código a través de interfaces gráficas, depuración de código entre otras muchas herramientas que iremos viendo.

Para poder instalar y configurar los plugin de Android en eclipse deberemos seguir los siguientes pasos:

1. En eclipse desplegar la pestaña de Help y presionar Install New Software

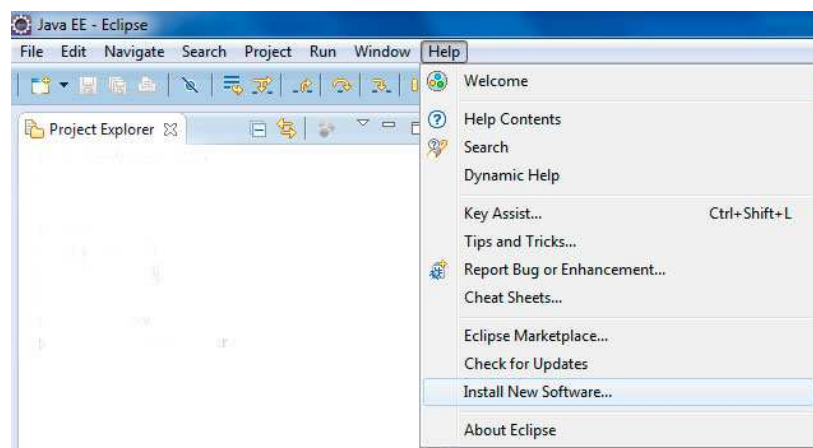


Fig. 13. Instalación del plugin de Android

2. En la nueva ventana que se abrirá pulsar el botón añadir (add) en la parte superior derecha.
3. En el cuadro de diálogo que se nos abrirá deberemos indicar el nombre de la ubicación que le queremos dar al repositorio (el nombre que queramos) y la dirección del repositorio que tiene que ser `https://dl-ssl.google.com/android/eclipse/` como se puede ver en la imagen.

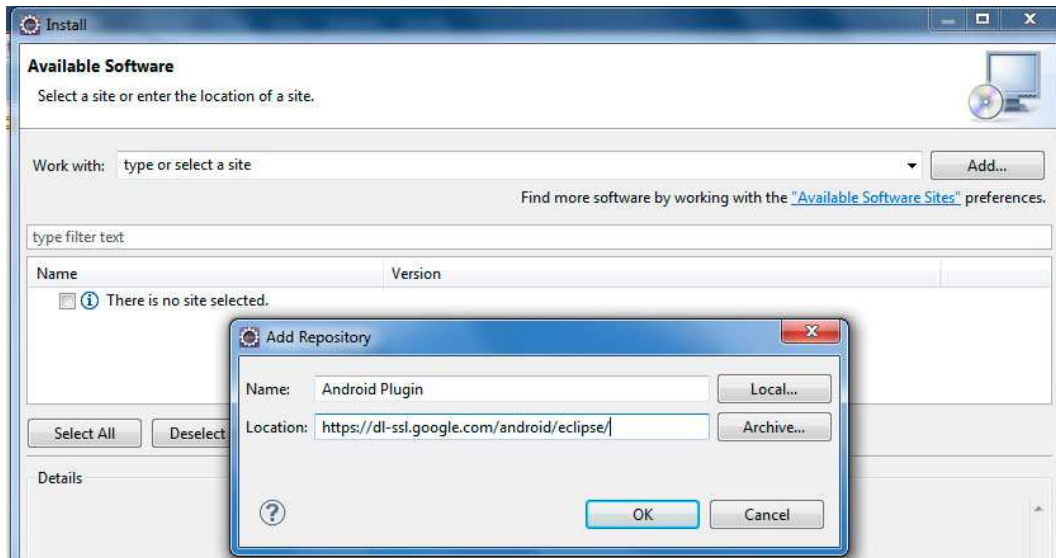


Fig. 14. Instalación del plugin de Android

4. Tras un breve período de tiempo nos indicara que plugins están disponibles, solo nos interesa Developer Tools ya que NDK Plugins son para programación en código nativo. Por lo tanto marcaremos solamente el primero, y presionamos Siguiente (Next)

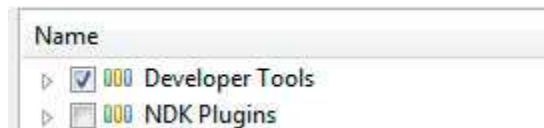


Fig. 15. Instalación del plugin de Android

5. En la siguiente ventana se mostrara una lista de las herramientas que van a ser instaladas, volvemos a pulsar Next.
6. En la última ventana aparecen los acuerdos de licencias, los cuales leeremos y aceptaremos y pulsaremos Finish.

Al terminar este proceso Eclipse nos pedirá reiniciarse, una vez hecho esto nos deberá aparecer nuestro espacio de trabajo, con el siguiente cambio.

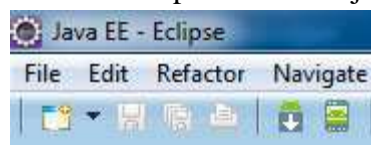


Fig. 16 Eclipse con los Plugins Android instalados.

Durante este proceso podemos encontrarnos con tres posibles problemas, el primero de ellos es que no reconozca la ruta del repositorio, en principio, se soluciona cambiando en la dirección https por http, aunque si el problema persistiera deberemos descargar los plugins desde <http://developer.android.com/sdk/installing/installing-adt.html> y añadir el archivo como en el paso 3.

El siguiente inconveniente que podemos encontrarnos será que al finalizar la instalación aparezca una ventana de advertencia de seguridad diciendo que la autenticidad o validez del software no se puede establecer, simplemente presionamos aceptar.

El último problema que se nos puede dar una vez reiniciado Eclipse es que no encuentre donde está instalado el SDK de Android automáticamente nos abrirá la venta de preferencia en la sección de Android donde indicamos donde está instalado en nuestro equipo el SDK, como podemos ver en la siguiente imagen.

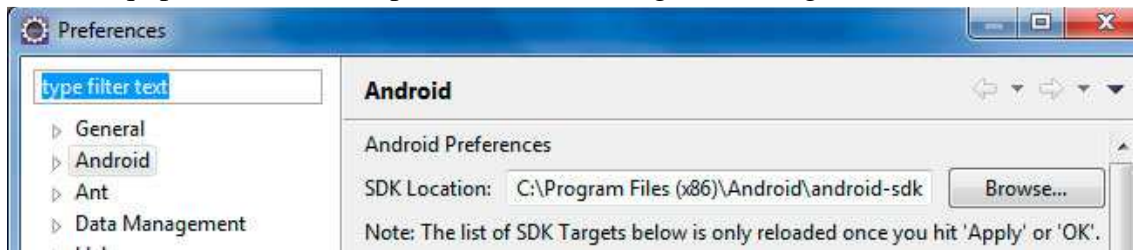


Fig. 17. Ventana para indicar la ubicación del SDK en eclipse

Y aunque Eclipse no nos avisara de que no encuentra el SDK, es bueno revisar que la ruta es correcta, lo podemos hacer manualmente accediendo a la ventana de preferencias, presionando en la pestaña Windows de Eclipse.

1.6 Ejecución y depuración.

Antes de presentar los elementos de un proyecto en Android, vamos a explicar cómo poder ejecutar y así poder depurar nuestras aplicaciones, para ello tenemos dos opciones, la primera de ellas sobre un terminal y la segunda sobre una terminal emulada en el mismo equipo.

1.6.1 Preparación de un terminal Android para la depuración y ejecución de aplicaciones.

En terminales con sistema operativo inferior al 4.0 para configurar el dispositivo tendremos que acceder Ajustes → Aplicaciones → Desarrollo y llegaremos a la imagen inferior.



Fig. 18 Ventana de Desarrollo de una tablet con Android 3.2

Aparecerán solamente tres opciones la primera sirve para habilitar o deshabilitar la depuración por USB, la segunda para permitir el apagado de la pantalla y la última para indicar ubicaciones falsas de GPS.

Si el terminal utilizado dispone de Android 4.0 o superior deberemos acceder mediante Ajustes → Opciones de desarrollador.

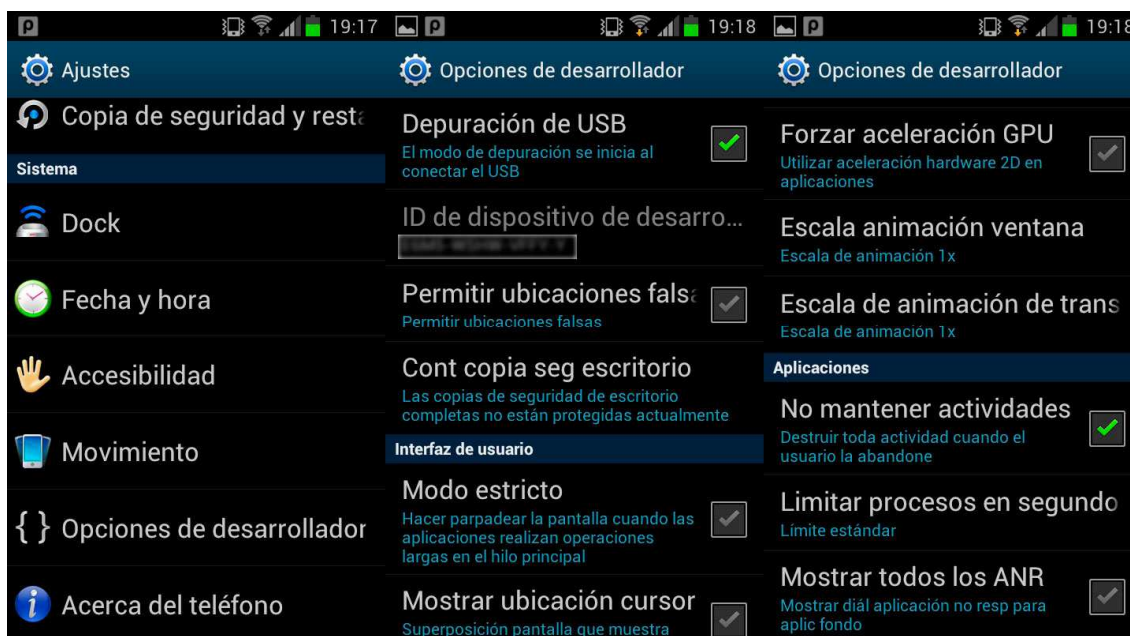


Fig. 19 Ventana de Desarrollo de un terminal con Android 4.0.3

Además de las opciones disponibles en las versiones anteriores se han añadido gran cantidad de opciones como ver el rendimiento de la CPU, hacer parpadear la pantalla cuando la aplicación realiza operaciones largas, entre otras.

1.6.2 Emulación de un terminal Android para la depuración y ejecución de aplicaciones.

Durante la instalación del SDK Manager, también se instaló Android Virtual Device Manager (AVD Manager), una herramienta que nos permitirá la creación de máquinas virtuales Android, para poder acceder AVD Manager tenemos dos opciones, una de ellas a través de eclipse el segundo icono que observamos en la Fig. 15 y la segunda opción dependerá del sistema operativo en el caso de Windows el acceso directo en la barra inicio como podemos observar en la Fig. 9

Una vez localizado, la primera ventana nos presenta las máquinas virtuales creadas hasta el momento, si es la primera vez que lo abrimos aparecerá vacía como vemos en la siguiente imagen además en la parte derecha de la imagen podemos ver las

diferentes opciones que nos ofrece AVD Manager; crear una nueva máquina, modificarla, borrarla, repararla, ver su estado actual e iniciarla.

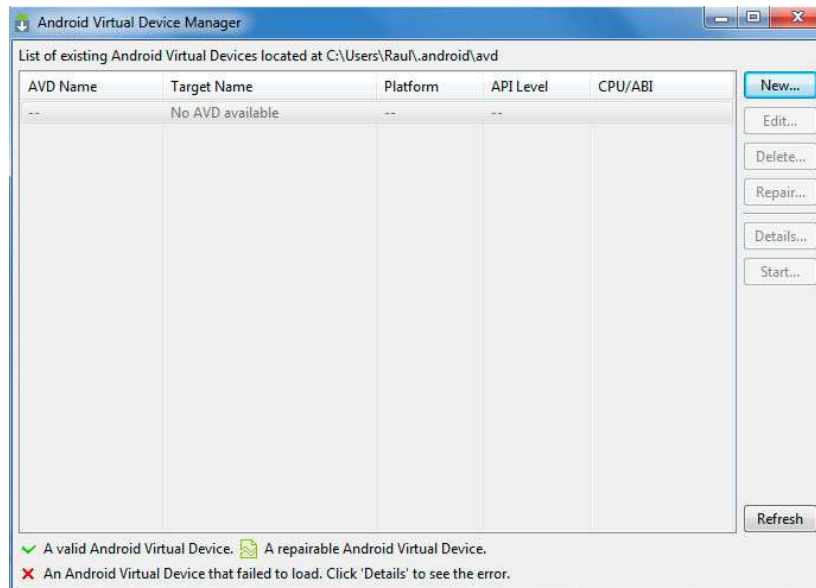


Fig. 20. AVD Manager

Para crear una nueva máquina virtual pulsaremos el botón New... el cual abrirá el siguiente diálogo.

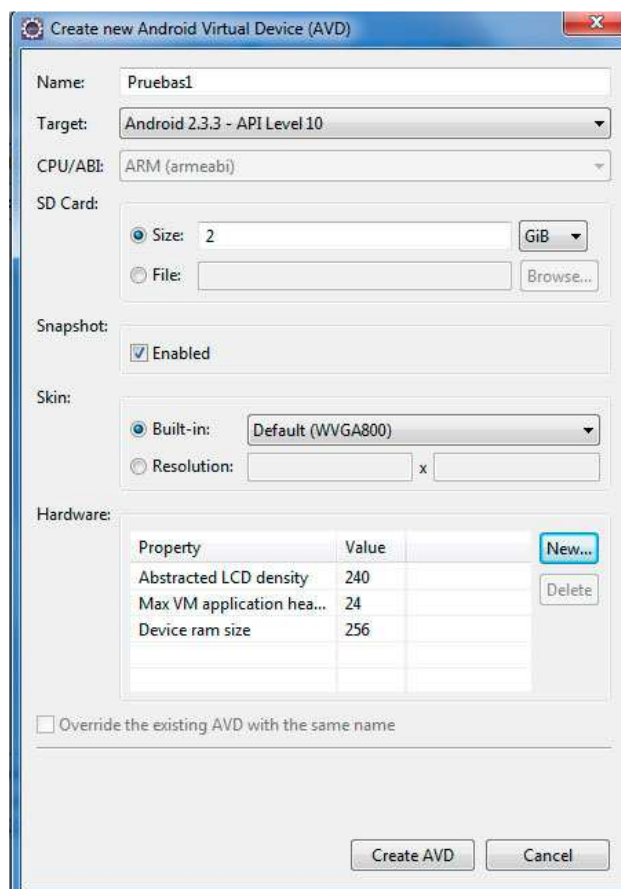


Fig. 21. Creación de una nueva máquina virtual mediante AVD Manager

En el tendremos que rellenar el nombre que deseamos darle a nuestra máquina, indicarle la versión del sistema operativo, si tuviera diferentes CPUs disponibles elegir entre ellas e indicar el tamaño de almacenamiento del dispositivo mediante su SD Card.

Además tenemos una casilla para activar o desactivar *Snapshot* el cual nos permitirá hacer capturas de la ejecución al momento del cierre de la máquina virtual para poder iniciarla posteriormente de una manera más rápida.

Para terminar podemos añadir a la emulación mas componentes hardware, aparte de los que vienen por defecto o incluso modificar los valores de los actuales como aumentar la memoria RAM que asigna al dispositivo. Para añadir deberemos pulsar el botón New... se nos abrirá la siguiente ventana.



Fig. 22. Añadir herramientas a una maquina virtual Android

Desplegando la pestaña de Property podemos ver las diferentes opciones que se pueden añadir, en este caso vamos a darle a nuestro dispositivo dos botones uno para volver atrás y otro para volver al inicio, una vez decidido se pulsa OK y ya estará disponible en nuestra maquina virtual.

Una vez finalizado todo el proceso tendremos que pulsar *Create AVD* en ese momento tendremos disponible una nueva máquina y así aparecerá indicado en AVD Manager.

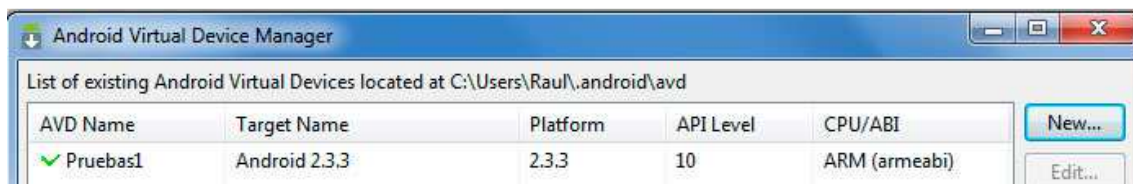


Fig. 23. AVD Manager con una nueva máquina virtual.

Finalmente para poner en marcha la máquina recién creada solo tendremos que pulsar el botón *start* teniendo seleccionada la máquina recién creada y dependiendo de la máquina donde esté siendo emulada tardará alrededor de un minuto en iniciarse por primera vez. Esta será la vista que veremos una vez haya terminado de arrancar.

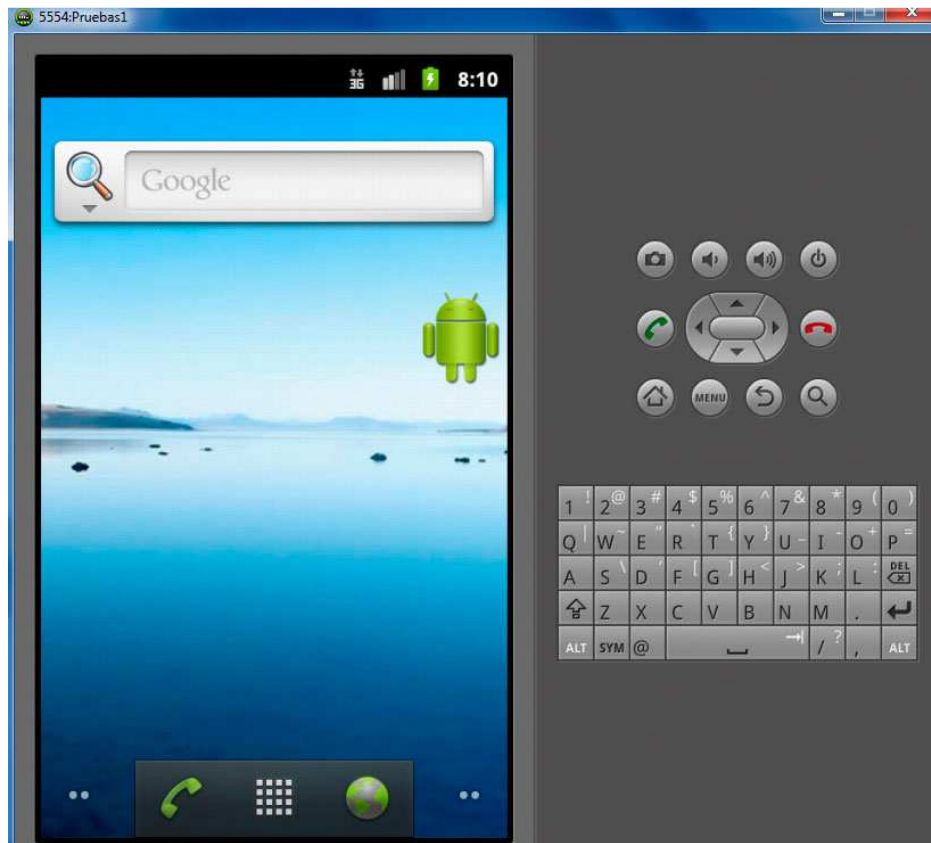


Fig. 24. Máquina virtual Android en ejecución.

Ya tenemos nuestra máquina virtual disponible podemos interactuar con ella, como si fuera un dispositivo real.

Capítulo II. Creación de aplicaciones Android.

A lo largo de esta memoria se presentará como funcionan e interactúan los diferentes elementos de las aplicaciones en Android mediante la creación de varias de estas.

2.1 1ª Aplicación. Números primos.

En este punto describiremos los pasos a seguir para la creación de una nueva aplicación (o proyecto Android), se realizará un sencillo ejemplo en el que se pedirá al usuario la introducción de un número y se comprobará si es primo o no indicando esto por pantalla.

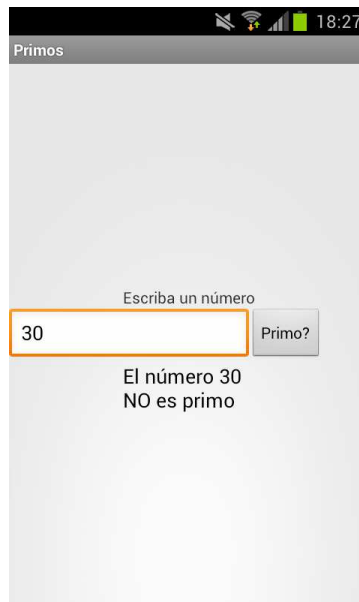


Fig. 25. Aplicación números primos.

Esa será la vista de nuestra aplicación una vez terminada, en este capítulo se explicará los elementos que componen un proyecto Android y como interactuar con ellos, además se explicará la clase *Activity* y su ciclo de vida y finalizaremos con unas simples técnicas para mejorar la depuración de nuestra aplicación y otros pequeños consejos.

Para iniciar nuestro nuevo proyecto se deberá iniciar eclipse y en la ventana de dialogo que se abrirá se indicará nuestro lugar de trabajo donde se almacenará todo nuestro trabajo.

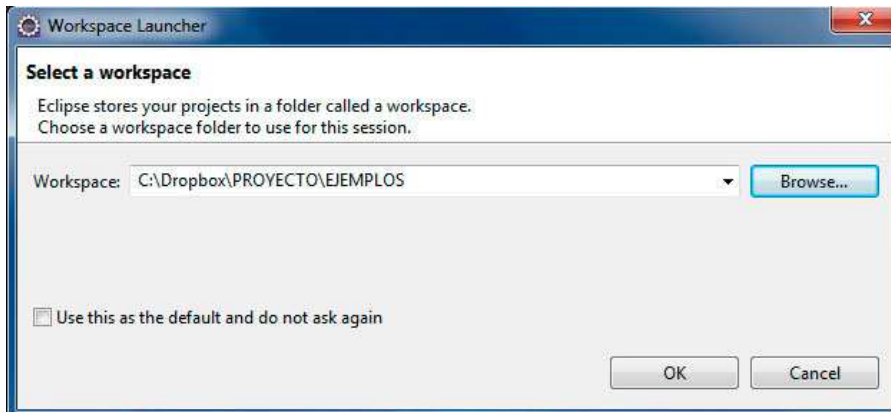


Fig. 26. Selección de nuestro espacio de trabajo.

Una vez que hemos indicado nuestro entorno de trabajo y tenemos abierto eclipse se pulsará Ctrl+N o en la barra de tareas File→New→Other en ese momento se abrirá la siguiente ventana.

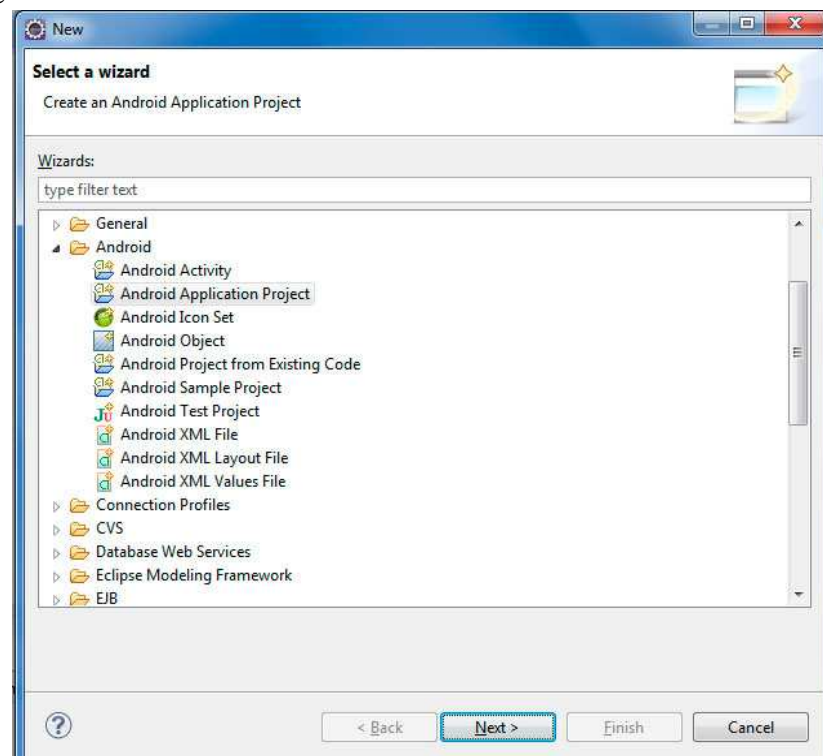


Fig. 27. Creación de un nuevo proyecto Android

En esta se seleccionará crear una *Android Application Project* y se pulsará siguiente, en la ventana que se abrirá se indicará el nombre de la aplicación, del proyecto y la versión del constructor y del sistema operativo mínimo requerido por la aplicación.

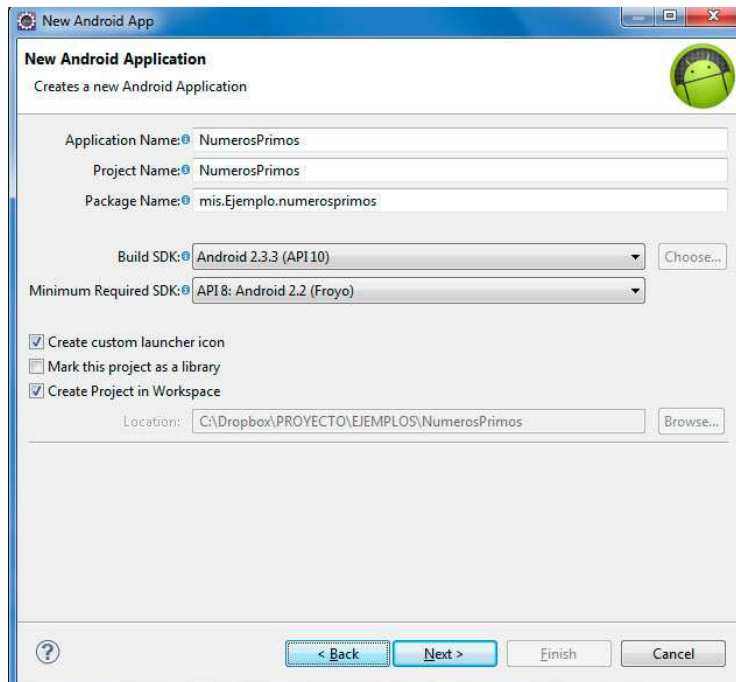


Fig. 28. Creación de un nuevo proyecto Android

Además a la vez marcado la casilla *Create custom launcher icon* (opción incluida en Junio de 2012 en la versión 20 de SDK tools) durante la creación del proyecto podremos crear los iconos de presentación de la aplicación, sino se marcará esta opción si quisiéramos personalizar los iconos se tendrá que hacer navegando a través de las carpetas del proyecto.

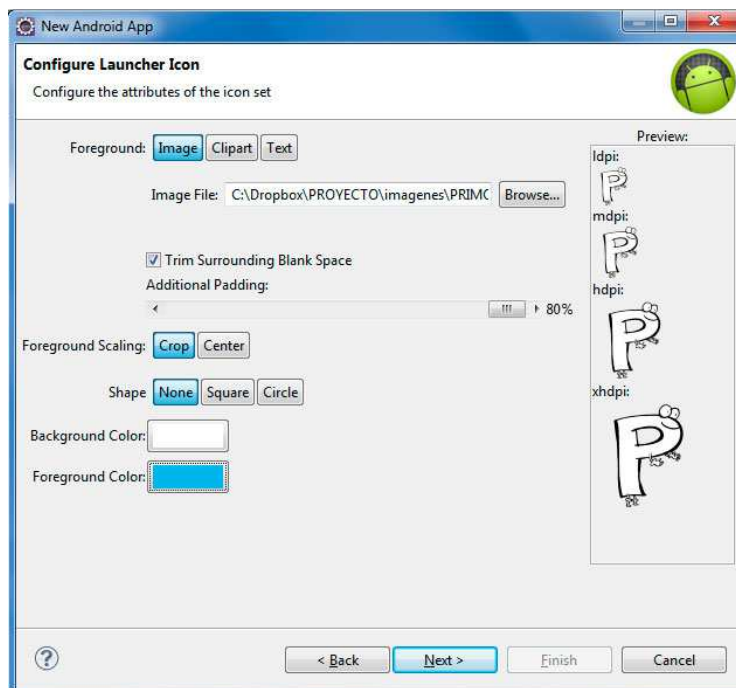


Fig. 29 Creación del icono de nuestra aplicación.

Al tener marcado la casilla de crear icono, la siguiente ventana que nos encontramos es la imagen superior, en ella podremos decidir si el icono de nuestra aplicación, es una imagen, clipart o texto entre otras opciones, lógicamente se busca una imagen que se distinga del resto de aplicaciones y que sea representativa de nuestra aplicación, en este ejemplo se eligió una imagen de una P, este será el resultado que veremos una vez finalizado.

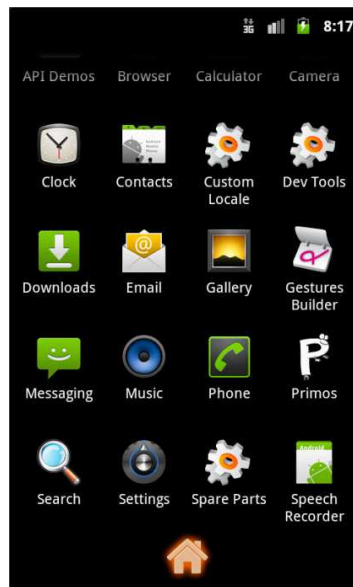


Fig. 30. Imagen final del icono en nuestro terminal.

Una vez finalizado el proceso de crear el icono, lo siguiente será decidir si nuestro proyecto tendrá o no alguna clase Activity nada mas crearse, lógicamente al igual que con el icono no es algo que no se pueda crear a posteriori, para este proyecto tan simple utilizaremos este automatismo.

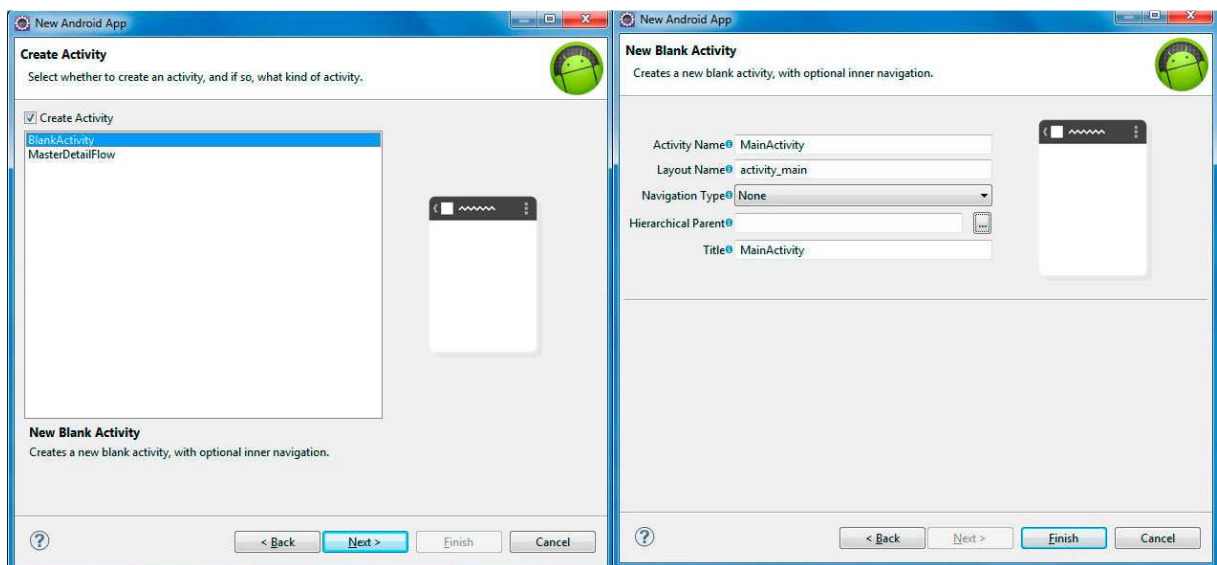


Fig. 31. Creación de la Activity principal.

Para ello simplemente marcaremos la casilla de *Create Activity* e indicaremos de los modelos existentes cual nos interesa, en la siguiente ventana podemos indicar el nombre de la clase, el nombre del archivo xml asociado, el tipo de navegación y el título.

Y aquí finalizaría todo este proceso tras el cual eclipse genera todo lo que nosotros hemos indicado preparando nuestro proyecto para que podamos empezar a moldear nuestra aplicación.

2.1.1 Elementos de un proyecto Android.

Cada proyecto Android se compone de una serie de elementos necesarios para la creación de una aplicación algunos de ellos deberán ser creados por el programador y otros serán autogenerados por eclipse, inicialmente el entorno de trabajo se compondrá de los siguientes elementos, que podremos ver desplegando la carpeta de nuestro proyecto, y de los cuales presentaremos con los que interactuaremos en nuestro proyecto.

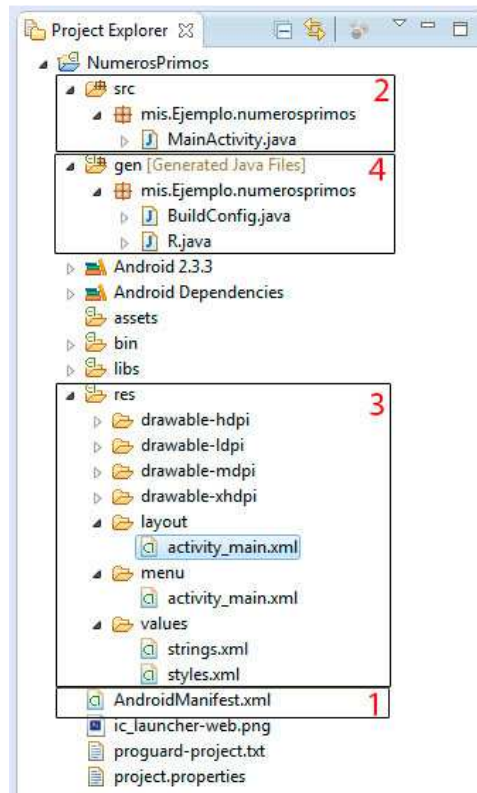


Fig. 32. Elementos de un proyecto Android.

1. **AndroidManifest.xml.** Presenta la información esencial sobre la aplicación a el sistema operativo, tal que este pueda correr el código de la aplicación, en el vendrá indicado lo siguiente:
 - El nombre del paquete Java de la aplicación que servirá como identificador único para la aplicación
 - Se describen los componentes de la aplicación, es decir que activities, services, broadcast receivers... componen la aplicación. Se nombraran a las clases que los implementan y publicará sus capacidades.
 - Se determinará qué proceso será el principal de la aplicación.
 - Se declara los permisos de la aplicación que deberá tener para acceder a partes protegidas de la API e interactuar con otras aplicaciones
 - Además de declarar que permisos tienen que tener las otras aplicaciones para interactuar con esta.
 - Se declara el nivel mínimo de la API que requiere la aplicación.

- Y las librerías necesarias.
2. **La carpeta src.** En ella se crearán todas las clases necesarias para nuestra aplicación.
 3. **La carpeta res.** En ella estarán todos los recursos de nuestra aplicación divididas en las siguientes carpetas.
 - **Carpeta drawable.** En ellas se almacenarán todas las imágenes de nuestro proyecto desde los iconos hasta los elementos gráficos de la aplicación.
 - **Carpeta layout.** Aquí se encontrarán los archivos xml correspondientes a cada clase Activity en ellos se podrá definir los elementos, ubicación, atributos y parte de su comportamiento que interactuaran con el usuario de la aplicación.
 - **Carpeta menu.** También está relacionada con una (o varias) Activity pero es totalmente opcional se utiliza para menús desplegables usando el botón de ajustes del terminal.
 - **Carpeta values.** En ella se cargan elementos específicos de la aplicación como cadenas de texto predeterminadas, estilos, colores...
 4. **La carpeta gen.** Esta carpeta contiene clases java autogeneradas por eclipse, la utilidad de estas poder vincular los elementos definidos en el lenguaje de marcas con los elementos java , durante la memoria se verá dicha relación.

Como se observa en la imagen existen otros elementos como la carpeta bin que contiene el archivo .apk que es el instalador de nuestra aplicación, el cual contiene toda la información. Además de otros archivos de configuración o autogenerados que por ahora no son necesarios estudiar.

Nuestra primera aplicación solo va tener una activity por lo tanto solo se tendrá un archivo xml asociado por lo tanto todos los archivos que se ven en la figura 31 serán los que compondrán nuestra aplicación.

2.1.2 Modificar el layout de una Activity.

A la hora de modificar el archivo xml que define cada layout los plugins de Android para eclipse nos ofrece un editor gráfico, el cual ayuda bastante a la hora de entender que se está realizando y ver cómo será el resultado final de este. Lógicamente también podemos modificar este de forma textual como cualquier otro archivo xml. Por norma general se crea a través de la interfaz gráfica y se añaden pequeños detalles modificando directamente el archivo xml.

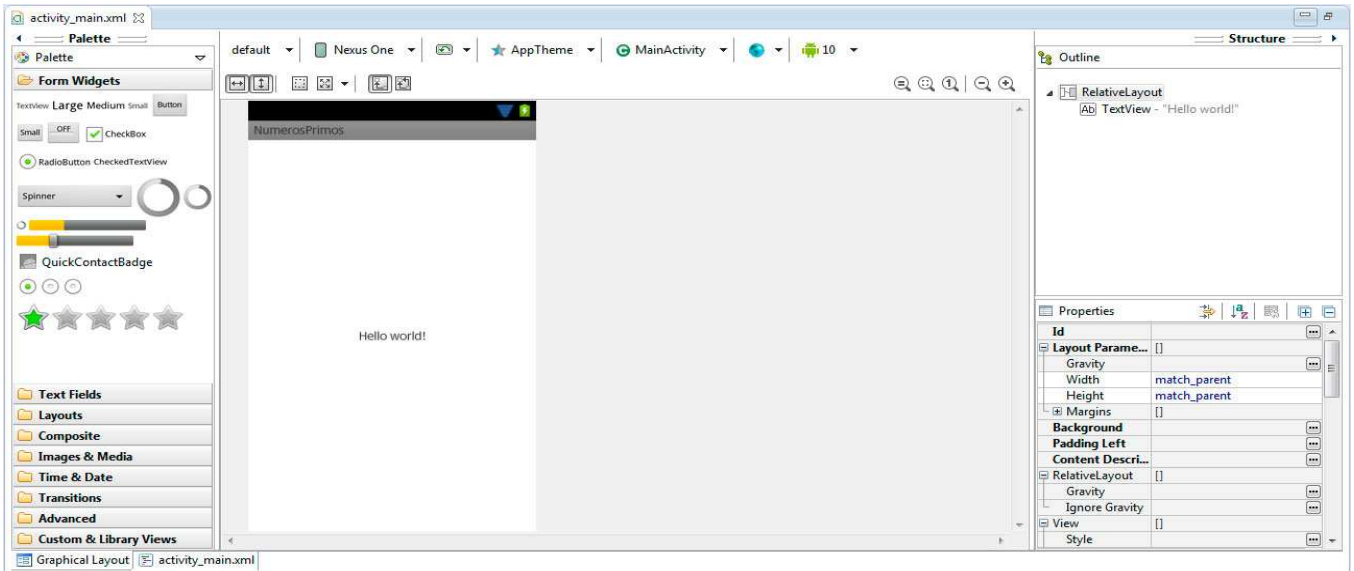


Fig. 33. Editor gráfico

Como se puede observar en la imagen superior se dispone de una paleta de herramientas en la parte izquierda divididas en carpetas con diferentes elementos, en la parte central de la imagen observamos el esqueleto de nuestro layout, superior a este observamos una barra de herramientas para modificaciones de estilo y versión, en la zona superior derecha observamos los elementos de nuestra aplicación y finalmente en la parte inferior derecha disponemos de las propiedades de cada elemento. Como se observa en barra inferior se puede cambiar la vista del editor gráfico por el textual.

Sobre el proyecto de los números primos vamos añadir los elementos que se veían en la figura 24, el primero de ellos un *TextView* que en este caso no abra que añadir podemos reutilizar el generado *Hello world!*, en la carpeta *Text Fields* se encuentran diferentes elementos para introducir texto en los cuales se puede limitar que tipo de texto se podrá introducir en este caso solo nos interesa números positivos, por lo tanto añadiremos la que nos indica que tiene esta función, además disponemos de un botón el cual está disponible en *Form Widgets* y finalmente se añade otro *TextView* para mostrar el resultado.

Se ajusta los elementos buscando una colación agradable sino termina de convencer la distribución obtenida se podrían añadir nuevos layout que contuvieran los elementos dando una nueva distribución, una vez terminado se observa el resultado en el editor textual.



Fig. 34 Creación de un layout.

La imagen superior es una instantánea del proceso de creación todavía no se añadido el botón pero ya se puede ver el código generado (o el código que ha generado la vista de la izquierda) además si se observa el código se ve que tenemos una advertencia y es que se esta asignando un valor de tipo string directamente cuando se debería haber utilizado un string creado para este tipo de uso para este caso lo que se hará será borrarlo ya que su valor dependerá de la ejecución.

Cada elemento se compone de sus atributos, el atributo id servirá de identificador más adelante, el resto son atributos de diseño colocación, tamaño, apariencia...

Lo siguiente será modificar los string de diseño de la aplicación para que se adapten al uso de esta aunque se pueden modificar en un entorno gráfico sencillo en esta caso trabajaremos sobre el archivo xml.

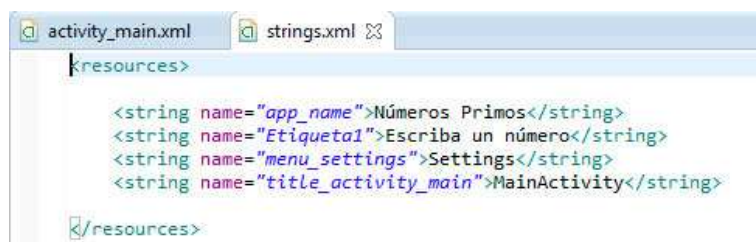


Fig. 35 Modificación de String

Como se ve en la imagen cada elemento se compone de dos campos su nombre (*name*) que hará de identificador y su valor. La manera de utilizarlos, por ejemplo, en un *TextView* es escribiendo dentro del campo *android:text* lo siguiente "*@string/Etiqueta1*".

Una vez terminado se tendría el esqueleto buscado para nuestra aplicación de los números primos pero todavía no hemos desarrollado la funcionalidad solo la apariencia, la cual se desarrolla en los .java.

2.1.3 La clase Activity

Es un concepto de programación único de Android, por norma general, en el desarrollo de aplicación en otros lenguajes hay un método estático principal (int main () {...}) que se ejecuta al iniciar la aplicación. Con Android, sin embargo, el funcionamiento es diferente, las aplicaciones pueden ser lanzadas a través de cualquier activity registrado en una aplicación (en el archivo AndroidManifest.xml), pero, por norma, siempre se especifica cuál de las activity será el punto de partido de nuestra aplicación no obstante podemos permitir que otras activity sean el punto de partida cuando la aplicación se bloquea o es cerrada por el sistema operativo, aunque en nuestro primer ejemplo no se verá más adelante presentaremos esta funcionalidad.

2.1.3.1 El ciclo de vida de activity.

El ciclo de vida de un activity comienza con su instalación y termina con su destrucción e incluye muchos estados durante ese tiempo. Cuando un activity cambia de estado, el método correspondiente del ciclo de vida se llama, notificando al correspondiente activity del cambio de estado inminente y permitiéndole ejecutar código con el fin de adaptarse a ese cambio.

A lo largo de esta capítulo se explicará qué función y que comportamiento tienen cada uno de los estados para poder desarrollar una aplicación de manera fiable y que se comporte de forma adecuada.

Una activity se compone de una serie de métodos que el sistema operativo llama a lo largo del ciclo de vida del mismo. Estos métodos implementan las funcionalidades necesarias para satisfacer los estados y la gestión de los recursos de sus aplicaciones, hay que ser cuidadoso a la hora de implementar dichos métodos porque en caso de no hacer esto de forma correcta podremos provocar inestabilidades no solo en nuestra aplicación sino en el sistema operativo subyacente.

2.1.3.2 Estados y métodos de activity.

Todos los activity utilizan una pila ofrecida por el sistema operativo Android en la cual el proceso que este interactuando con el usuario (visible en la pantalla) es el que se encuentra primero en la pila de procesos, siendo por normal general el siguiente aquel que haya sustituido al que entró en la primera posición, Android intenta mantener los procesos en ejecución lo máximo posible, pero no indefinidamente, ya que los recursos del sistema son limitados.

En base al estado en que estado se encuentre una determinada actividad, Android asignará una cierta prioridad, gracias a este se detectará aquellas actividades que no están en uso, permitiendo la recuperación de recursos asignados.

En función de su estado podemos distinguir esencialmente cuatro estados:

- **En funcionamiento o ejecución.** Se encuentra en el primer plano de la pantalla, esta interactuando con el usuario, como se dijo se encuentra en la parte superior de la pantalla.
- **En segundo plano.** La actividad ha perdido el foco pero sigue siendo visible, es decir, una ventana emergente, que por ejemplo, pregunta alguna información relacionada con la que se está ejecutando, provoca que nuestra actividad pasé a un segundo plano de manera temporal. Hay que tener en cuenta puede ser eliminada por el sistema en situaciones extremas de poca memoria.
- **No visible.** La actividad está totalmente oculta por otra actividad, se detendrá. Todavía conserva toda la información de su estado y puede el usuario volver a ellas estando en el estado en que la abandono. Es más fácil que Android elimine de memoria estas actividades cuando necesite asignar nuevos recursos.
- **En pausa o detenida.** El sistema puede liberarla de memoria simplemente llevando la actividad al método correspondiente o matando el proceso. La clave es que cuando se muestre de nuevo al usuario está aparecerá en estado inicial y no en el que se encontraba cuando fue detenida.

El siguiente diagrama muestra los estados más importante de una activity, los cuadros rectangulares representan llamadas a métodos que pueden ser implementados para llevar acabo determinadas funciones sino se implementan se llamará al método de la superclase y los estados ovalados nos indican el estado de la aplicación desde un punto más global.

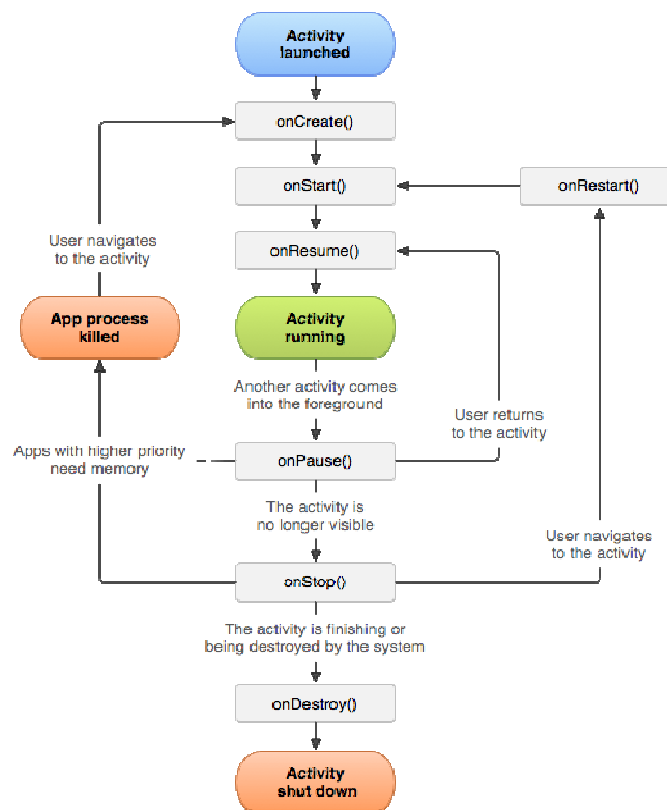


Fig 36. Ciclo vital de una actividad

Como podemos observar en la imagen existen 7 métodos diferentes en los cuales se encontrará una activity. En la siguiente tabla haremos una breve descripción de cada uno de ellos, además de indicar en qué punto el sistema operativo puede eliminar a la aplicación de la pila y finalmente indicaremos cual es su siguiente método.

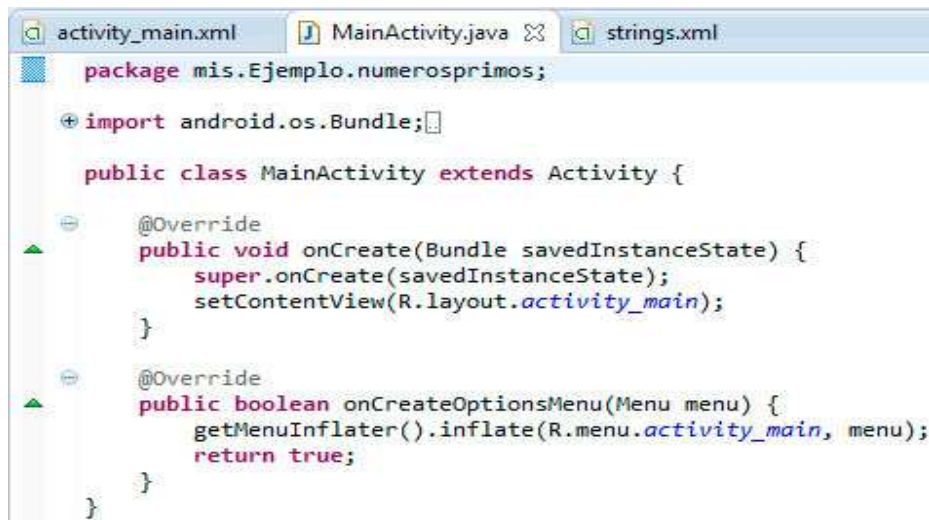
| Método | Descripción | ¿Eli.? | Siguiente |
|--------------------|---|--------|---------------------------|
| onCreate() | Se llama al crear la activity. Una activity debe reemplazar este método para configurar su pantalla principal y realizar cualquier configuración inicial. El parámetro Bundle de entrada nos servirá para pasar información a nuestra activity en su creación. | No | onStart() |
| onRestart() | Este método se llama después de que la activity se haya detenido, antes de que se inicie de nuevo y siempre va seguido de un método start, es utilizado en situación como reinicio de la aplicación desde el administrador de tareas de Android. | No | onStart() |
| onStart() | Se llama antes de que la activity interactúe con el usuario. Se deberá reemplazar si se necesita realizar cualquier tarea justo antes de que se haga visible. | No | onResume() o onStop() |
| onResume() | Es llamado cuando vuelve a interactuar con el usuario después de un estado de pausa. | No | onPause() |
| onPause() | Se utiliza antes de poner la activity en segundo plano. En él se almacenarán los datos que se quieran guardar y se destruirán los objetos para liberar recursos. La implementación de este método deben finalizar lo antes posible, ya que ningún activity sucesivo se reanudará hasta que este método devuelva un valor. | No* | onResume() o onStop() |
| onStop() | Es invocado cuando la activity ya no es visible por el usuario, ya que otro activity se ha iniciado aunque también es utilizado cuando se realiza un cambio de orientación más adelante se verá sus posibles usos. | Si | onRestart() o onDestroy() |
| onDestroy | Es el último método antes de que sea destruido, después de llamar a este método la activity se eliminará el sistema operativo destruirá permanentemente los datos de la activity. | Si | Ninguno |

*Depende de la versión del sistema operativo, en versiones anteriores a la 3.0 no se puede eliminar en ese estado

Además de los métodos descritos y mostrados en el diagrama se proporciona otro método, **onSaveInstanceState(Bundle bundle)** es utilizado para dar la oportunidad para guardar los datos cuando se produce un cambio, por ejemplo, un cambio de orientación de pantalla.

El método *onCreate* estará presente en todas las activity por norma general y en los primeros ejemplos que veamos será el único que inicialmente este implementado según se avanza en el desarrollo de aplicaciones el uso de los otros métodos será algo muy común.

Como se explico, se puede encontrar la actividad de nuestro proyecto dentro de la carpeta src que cuelga del raíz, con el nombre que se asigno en la creación MainActivity.java y ya se nos presenta con un pequeño esqueleto.



```
activity_main.xml | MainActivity.java | strings.xml
package mis.Ejemplo.numerosprimos;

import android.os.Bundle;

public class MainActivity extends Activity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

Fig. 37. Estado inicial de una actividad

Lógicamente nuestra clase hereda las propiedades de la clase Activity, además ya nos ha sobrescrito dos métodos *onCreate*, ya descrito, y *onCreateOptionsMenu(Menu menu)* que es utilizado para desplegar un menú de opciones al pulsar el botón de ajustes.

Lo primero que se realizará será añadir un método que se le pase por parámetro un número entero y nos devuelva un booleano indicando si el número es primo o no (un número natural es primo si solamente si es divisible por 1 y el mismo). Y aunque en principio el método aceptaría números negativos, nuestro *EditText* definido en el archivo *activity_menu.xml* permitirá solo introducir números naturales. Por lo tanto el método quedaría de la siguiente manera.

```
public boolean esPrimo(int numero){
    int contador=2;
    boolean primo = true;
    while((primo)&&(contador!=numero)){
        if(numero%contador == 0)
            primo=false;
        contador++;
    }
    return primo;
}
```

Fig. 38. Método para saber si un número es primo

Una vez que disponemos del método `esPrimo` en nuestra actividad añadiremos como en cualquier clase los atributos que tiene, en nuestro caso se tendrán tres:

```
public class MainActivity extends Activity {  
  
    private Button miBoton;  
    private EditText entrada;  
    private TextView salida;  
  
}
```

Fig. 39. Atributos de la clase `MainActivity`

El elemento `Button` se encargara de la gestión del botón, `EditText` se encargará de la gestión de la lectura de los datos escritos por el usuario y `TextView` se usará para mostrar los resultados. Por lo tanto la modificación del método `onCreate` será la siguiente.

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    entrada = (EditText) findViewById(R.id.editText1);  
    salida = (TextView) findViewById(R.id.textView2);  
  
    miBoton=(Button) findViewById(R.id.button1);  
    miBoton.setText("Primo?");  
  
    miBoton.setOnClickListener(new View.OnClickListener() {  
  
        public void onClick(View v) {  
            int num=Integer.parseInt(entrada.getText().toString());  
            if(esPrimo(num))  
                salida.setText("El número "+ num + " es primo");  
            else  
                salida.setText("El número "+ num + " NO es primo");  
        }  
    });  
}
```

Fig. 40. Implementación del método `onCreate`

Lo primero que hacemos es llamar al constructor de la superclase y pasarle un `Bundle` que se utiliza para pasarle información de una actividad a otra en este caso estará vacío, el método `setContentView(View)` añade la interfaz de usuario (UI, user interface) definida en el archivo xml correspondiente a la actividad y lo siguiente será relacionar los atributos de nuestra clase con los elementos de la IU para ello disponemos del método `View findViewById(Int)`

Una vez relacionados los elementos de xml con nuestro atributos lo siguiente será trabajar con ellos, lo primero que hacemos es dar un nombre a nuestro botón, es decir que aparezca sobre un etiqueta de texto, esto lo se podría realizar en la parte de diseño en los archivos xml pero Android permite esta dualidad a la hora del diseño de las aplicaciones.

Lo siguiente será atender el evento de pulsar el botón para ellos deberemos indicar que nuestro botón puede producir eventos y que será el mismo con el método

onClick(View v) el encargado de atenderlos, esta es una de las formas que podemos trabajar con botones aunque como veremos más adelante existen otras más sencillas y simples de trabajar con ellos aprovechando la dualidad antes citada.

Cada vez que se pulse el botón se leerá del *EditText* entrada, haciendo la correspondiente transformación a entero, y se comprobará que el número introducido es primo y en función del resultado se indicará por pantalla un mensaje otro utilizando el *TextView* salida como se ve en la imagen.

Ya tenemos nuestra aplicación terminada y aunque hemos utilizado elementos como los View los cuales se explicarán más adelante el siguiente paso será la puesta en marcha de nuestra aplicación.

2.1.4 Probar una aplicación desde Eclipse.

Ya preparamos los dispositivos tanto virtuales como reales para poder depurar aplicaciones es muy posible que en caso de no estén instalados los drives del dispositivo eclipse no lo detecte aunque si aparezca como almacenamiento de datos.

Para ello deberemos entrar en la página developer de Android y en la zona de herramientas, buscar en la barra izquierda Extras→USB Driver deberemos buscar la tabla con los OEM USB Driver que estará a mitad de la página web, como se ve en la siguiente imagen.

| OEM | Driver URL |
|-------------------|---|
| Acer | http://www.acer.com/worldwide/support/mobile.html |
| ALCATEL ONE TOUCH | http://www.alcatel-mobilephones.com/global/Android-Downloads |
| ASUS | http://support.asus.com/download/ |
| Dell | http://support.dell.com/support/downloads/index.aspx?c=us&cs=19&l=en&s=dhs&~ck=anavml |
| Foxconn | http://drivers.cmcs.com.tw/ |

Fig. 41. <http://developer.android.com/intl/es/tools/extras/oem-usb.html#Drivers>

Como se observa cada fabricante ofrece los drivers para sus dispositivos en sus páginas web, una vez descargado e instalado es muy probable que se requiera reiniciar el sistema.

Una vez que todo está preparado, en eclipse pulsamos sobre la flecha que hay junto al icono de ejecución como se ve en la siguiente imagen y entramos en *Run Configurations...*

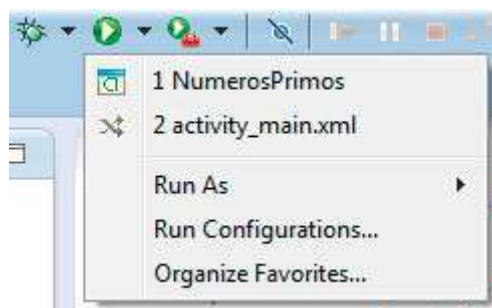


Fig. 42. Menú de ejecución de eclipse.

En la ventana que se desplegará deberemos indicar que cada vez que se ejecute una aplicación eclipse pregunte en que dispositivo se desea instalar, para ello se debe marcar la casilla *Always prompt to pick device*.

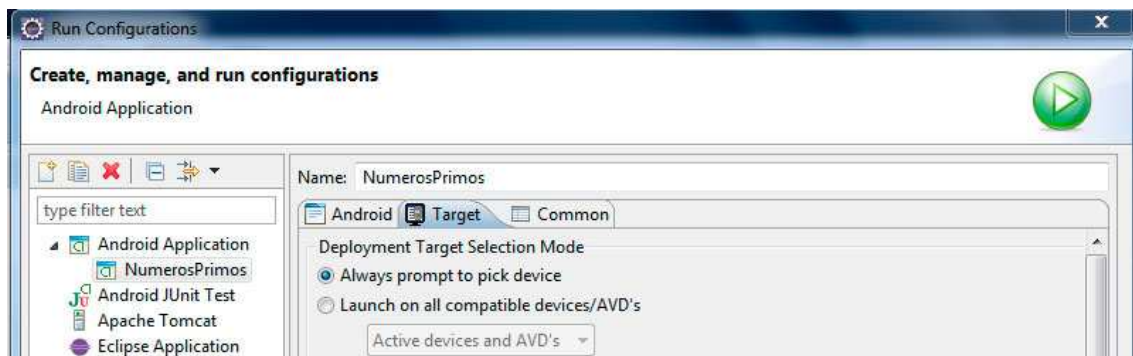


Fig. 43. Run Configurations

Cuando pulsemos el botón de Run por primera vez Eclipse nos preguntara que como deseamos ejecutar nuestra aplicación para ello deberemos marcar la opción Android Application.

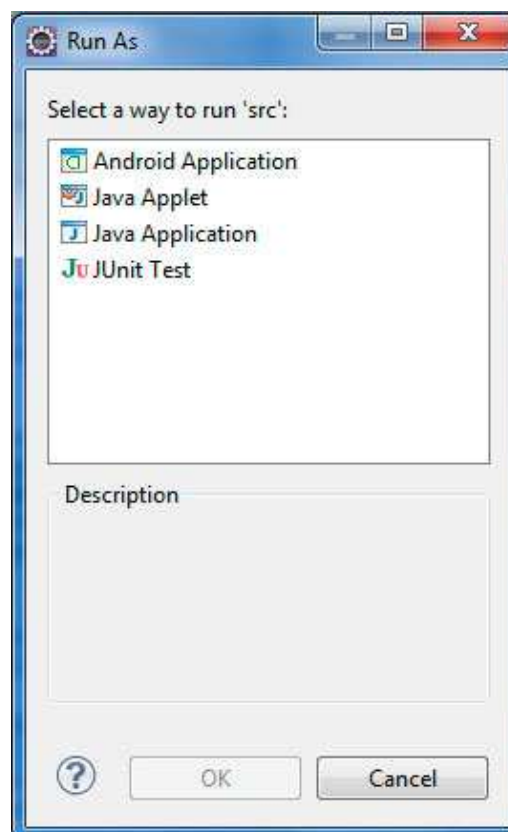


Fig. 44. Run As

Una vez elegida la opción nos preguntara en que dispositivo deseamos ejecutar la aplicación como se ve en la imagen.

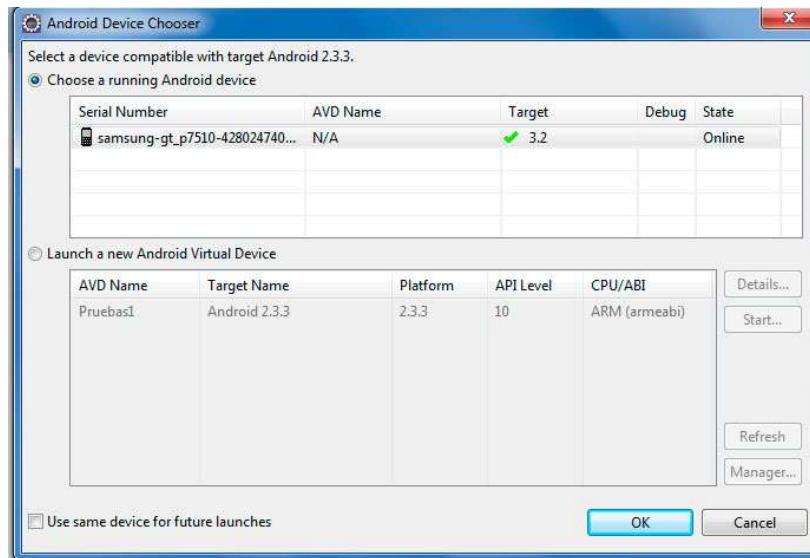


Fig. 45. Elección del dispositivo de ejecución de una Aplicación Android

Lo útil de realizar la instalación de una aplicación de esta manera es que eclipse desinstala la versión existente, instala la versión actual y la ejecuta, haciendo el proceso que se tendría que hacer de forma manual si los instalásemos desde apk.

Hay que tener en cuenta que si detecta una versión instalada desde el apk deberemos de eliminarla de forma manual ya que no nos dejara probar una nueva para proteger la existe.

En nuestro caso vamos a probar la aplicación en una tablet que es el dispositivo que se ve en la imagen superior, este sería la vista final de nuestra aplicación.

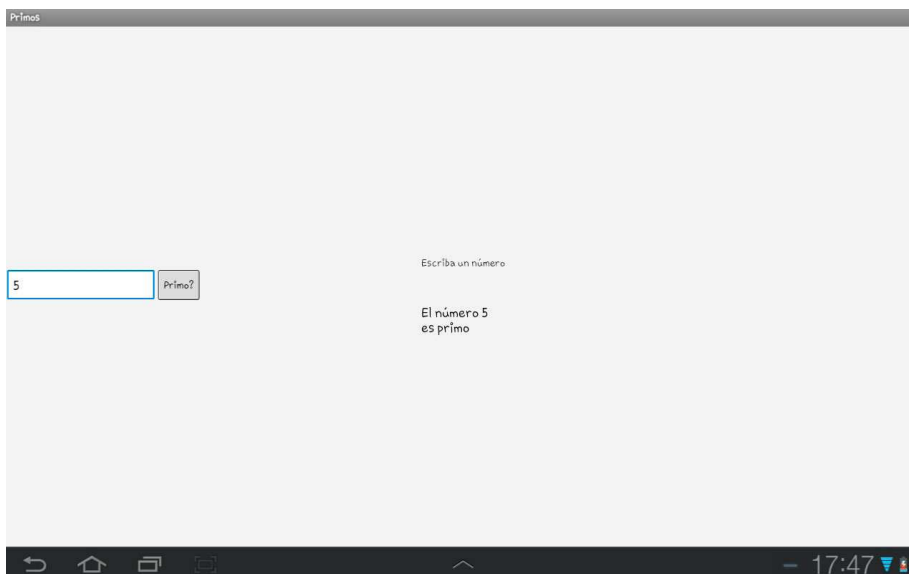


Fig. 46. Imagen de la aplicación números primos en una tablet.

Como se puede observar la imagen de la aplicación no es la que nosotros habíamos diseñado con todo en el centro de la imagen, eso es debido a que nuestro diseño además de no ser del todo correcto está preparado para terminales con un tamaño de pantalla mucho menor.

2.1.5 Depuración de aplicaciones con Eclipse

Otra de las herramientas que se instalan con el SDK Manager y que se integra en eclipse con los plugin de Android es la herramienta DDMS (Dalvik Debug Monitor Server), con ella se puede visualizar los hilos donde se ejecuta la aplicación, el uso de la memoria, el rendimiento de la CPU, el consumo de datos de red...

Para poder ver todo esto se debe acceder a la vista DDMS de eclipse, en la parte superior derecha se observa inicialmente en la vista Java EE, junto a el hay un cuadro que si pasamos el ratón por encima nos indica su utilidad *Open Perspective*, también se puede acceder usando la barra superior entrando en Windows→Open Perspective→Other y se desplegará la siguiente ventana.

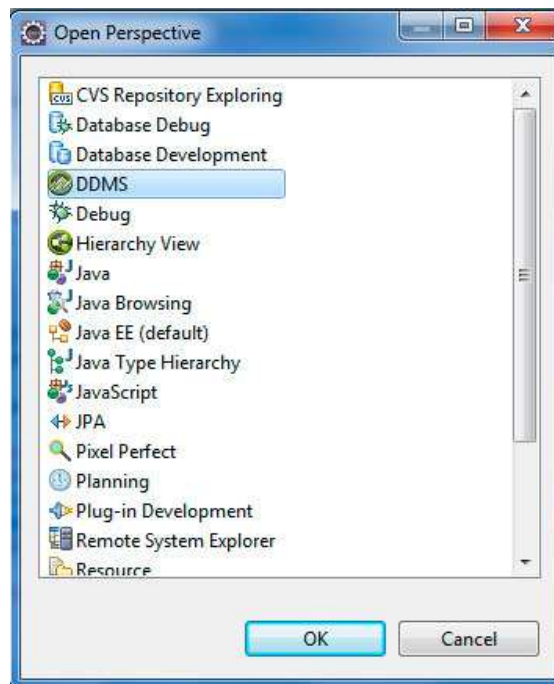


Fig. 47. Open Perspective

En ella se elegirá la perspectiva DDMS, para cuando se quiera volver a la vista anterior simplemente en la zona superior derecha aparecerán las vistas usadas recientemente.

La nueva ventana se divide en tres zonas, la primera de ellas es la pestaña de *Devices*, en la cual se pueden ver todos los dispositivos disponibles tanto virtuales como físicos, además de una serie de herramientas para aplicar a la aplicación que se esté probando; la segunda zona que se observa nos permitirá ver los hilos que utiliza nuestra aplicación, ver la memoria RAM que consume, ver el consumo de los datos de la redes,

un explorador de archivos y una herramienta para simular llamadas entrantes, envió de sms, localizaciones GPS...; y finalmente en la zona inferior tenemos la herramienta *LogCat* en el que se puede visualizar todos los mensajes que pueden emitir los diferentes procesos, y filtrarlos según nos interese, además esta herramienta también es visible desde la perspectiva de desarrollo, en el siguiente punto veremos cómo generar mensajes para poder hacer una mejor depuración de nuestra aplicación.

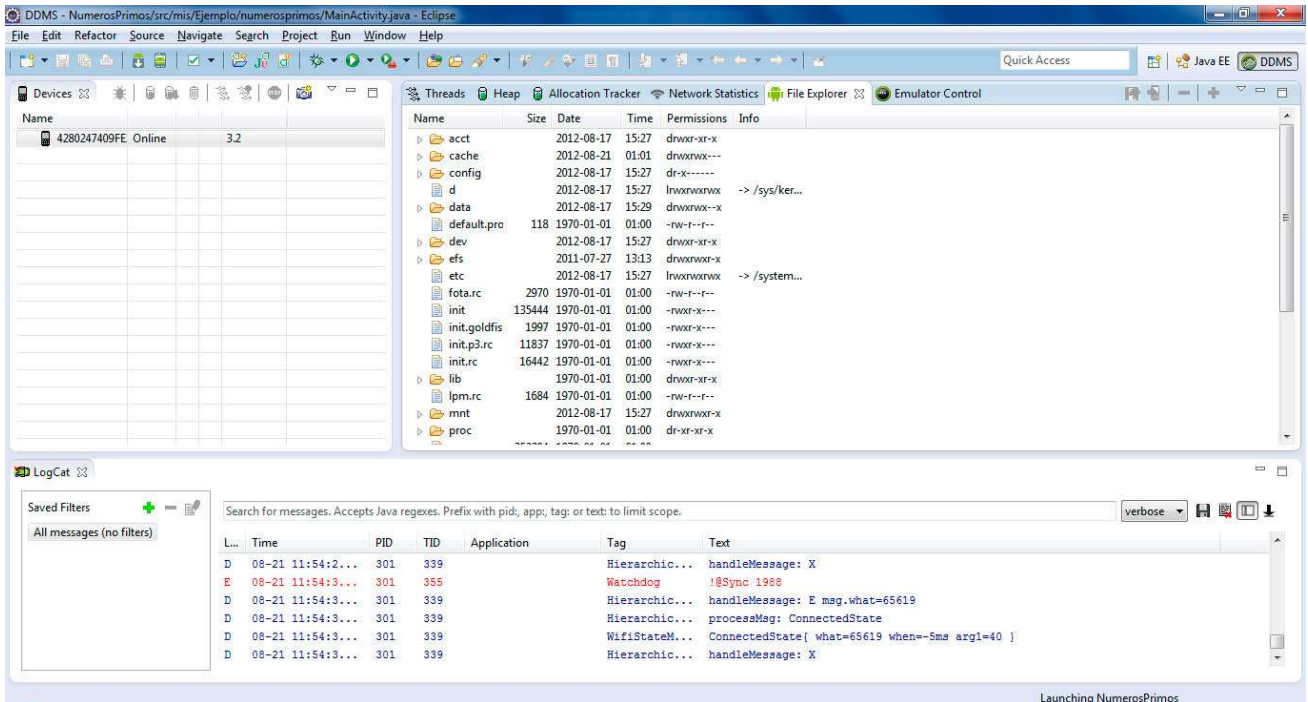


Fig. 48. Perspectiva DDMS.

Para aplicaciones simples como la aplicación de los números primos no es necesaria tanta información pero esta herramienta será muy útil cuando el proyecto a realizar tenga muchos procesos y utilice bastantes recursos del sistema, aunque la herramienta *LogCat* si será muy útil a la hora de depurar errores en cualquier proyecto.

Para poder lanzar mensajes disponemos de la clase log definida en el paquete `android.util.log` la cual nos ofrece diferentes métodos para lanzar mensajes, en la siguiente imagen vemos como se han añadido el envío de tres mensajes.

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.i(tag, "Estado 1");
    entrada = (EditText) findViewById(R.id.editText1);
    salida = (TextView) findViewById(R.id.textView2);

    miBoton=(Button) findViewById(R.id.button1);
    miBoton.setText("Primo?");
    Log.i(tag, "Estado 2");

    miBoton.setOnClickListener(new View.OnClickListener() {

        public void onClick(View v) {
            Log.d(tag, "Se pulso el botón");
            int num=Integer.parseInt(entrada.getText().toString());
            if(esPrimo(num))
                salida.setText("El número "+ num + " es primo");
            else
                salida.setText("El número "+ num + " NO es primo");
        }
    });
}

```

Fig. 49. Números primos con depuración usando log.

En la imagen se pueden ver que hemos usados dos métodos diferentes, aunque el funcionamiento va ser el mismo supuestamente son tipos de mensajes diferentes, existen muchos más, como cuando se produce una excepción, los cuales se puede leer en la página <http://developer.android.com/intl/es/reference/android/util/Log.html>

El prototipo de los métodos es *public static int i (String tag, String msg)* donde la etiqueta tag es recomendable que sea común en todos los mensajes, y que en nuestro ejemplo hemos definido como *numerosPrimos*, y la etiqueta msg que será el mensaje que será lanzado.

Una vez añadidos los mensajes que se desea que lance nuestra activity lo mejor es añadir un filtro para que solo se muestren estos, para ello en la ventana inferior seleccionaremos la pestaña de *LogCat* y crearemos un filtro nuevo.

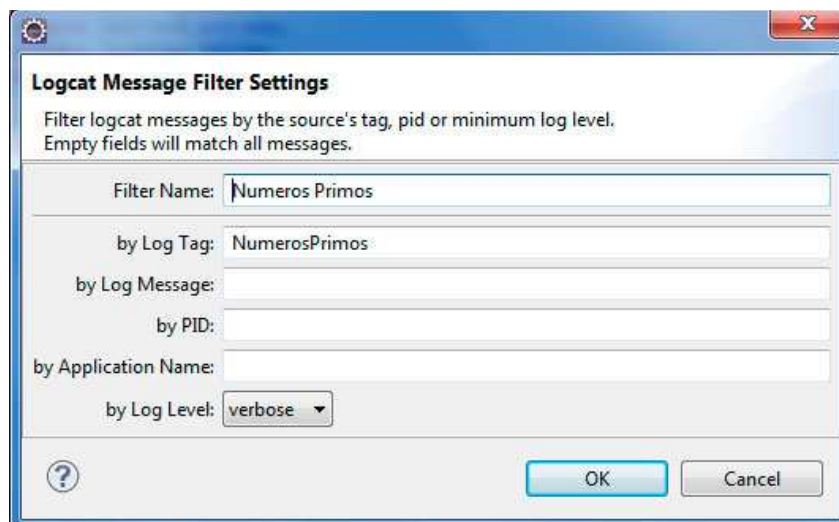


Fig. 50. Filtrado de los mensajes de Logcat

En ella indicaremos el nombre del filtro y qué condiciones deben cumplir los mensajes para que sean mostrados en *LogCat* en este caso utilizaremos la herramienta depurar por Tag con el valor *NumerosPrimos*, ahora al depurar nuestra aplicación nos mostrará lo siguiente.

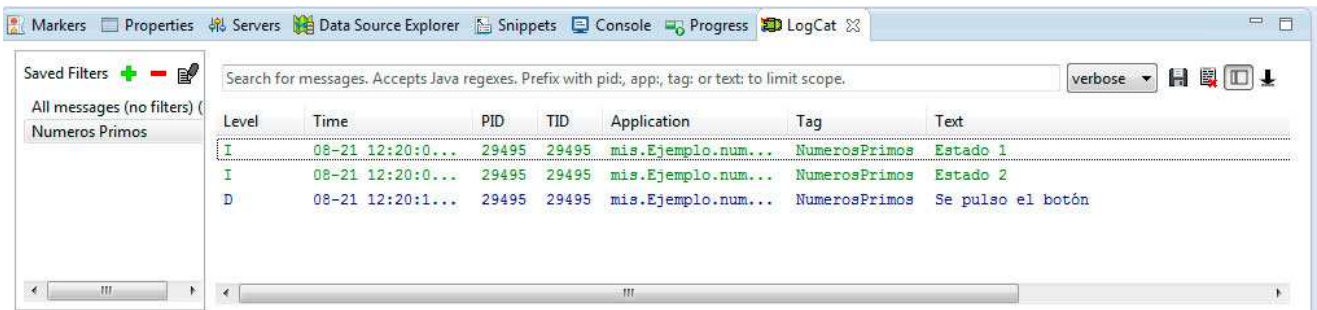


Fig. 51. Muestra del LogCat con un filtro

Los mensajes se mostrarán por orden de ejecución, mostrando su tipo, momento de ejecución, identificador del proceso, aplicación de origen, su Tag y el mensaje que les acompaña. Gracias a esto podremos descubrir errores o puntos muertos en nuestra aplicación

2.1.6 Mejora de aplicaciones.

Ya esta puesta en marcha la aplicación números primos ahora se buscará añadir pequeñas mejoras, que se utilizan bastante a menudo y que mejoran el resultado final tanto a nivel de programación como de presentación al usuario.

2.1.6.1 Gestión del evento de pulsar un botón.

Hasta ahora la única forma que se ha presentado la forma de atender un evento de pulsar un botón en Android era a través del código Java. Existe otra manera más sencilla y es indicando en el archivo XML que método atenderá el evento de pulsación del botón, no habrá que añadir un atributo botón en la clase Activity, ni indicar que producirá un evento simplemente implementar el método que se indique en el XML.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.i(tag, "Estado 1");
    entrada = (EditText) findViewById(R.id.editText1);
    salida = (TextView) findViewById(R.id.textView2);
    Log.i(tag, "Estado 2");
}

public void pulsado(View v){
    tarea();
}

public void tarea(){
    Log.d(tag, "Se pulso el botón");
    int num=Integer.parseInt(entrada.getText().toString());
    if(esPrimo(num))
        salida.setText("El número "+ num + " es primo");
    else
        salida.setText("El número "+ num + " NO es primo");
}

<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignTop="@+id/editText1"
    android:layout_toRightOf="@+id/editText1"
    android:text="@string/esPrimo"
    android:onClick="pulsado"/>
```

Fig. 52. Gestión de botones en Android

En la parte inferior derecha de la imagen superior se ve las dos modificaciones realizadas al botón para que una muestra una etiqueta en el interior como antes y se le ha añadido el atributo *onClick* en el cual indicamos que cada vez que haya una pulsación sobre el se atenderá en el método *pulsado*.

En la parte izquierda de la imagen vemos como queda la clase Activity ahora, por un lado tenemos el método que atiende cada vez que se pulsa y por otro lo que se hará cuando se pulse de esta manera podemos reutilizar el código para si queremos que otro evento realice la misma acción.

Finalmente podemos observar que el atributo *private Button miBoton* ha sido eliminado de nuestra clase Activity.

2.1.6.2 Generar y gestionar eventos con el teclado virtual

Es muy común que cuando se realiza un formulario como el de nuestra actividad de los números primos deseamos que cuando se pulse la tecla *enter* el comportamiento fuera el mismo que el del botón para ello deberemos cambiar el comportamiento del *EditText* para que cada vez que se pulse una tecla registra el evento, para ello se debe de incrementar lo siguiente en el método *onCreate*.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.i(tag, "Estado 1");
    entrada = (EditText) findViewById(R.id.editText1);
    salida = (TextView) findViewById(R.id.textView2);
    Log.i(tag, "Estado 2");

    entrada.setOnKeyListener(new OnKeyListener() {

        public boolean onKey(View v, int keyCode, KeyEvent event) {
            if(keyCode==KeyEvent.KEYCODE_ENTER && event.getAction()==KeyEvent.ACTION_UP){
                Log.d(tag, "Se pulso la tecla enter");
                tarea();
                return true;
            }
            else
                return false;
        }
    });
}
```

Fig. 53. Registro de evento de pulsación de teclas

Cada vez que se pulse una tecla esta será evaluada en el método *onKey*, en la variable *int keyCode* se almacena un valor diferente para cada tecla y en *keyEvent* el evento que acompaña a tecla.

Existen tres tipos de eventos para las teclas, uno que se esté pulsando, la segunda que se libere la pulsación y la tercera que se estén pulsando varias teclas simultáneamente.

En la actividad de números primos comprobaremos que la tecla pulsada corresponde con la tecla *enter* y que la pulsación termino para realizar la tarea. Para ello disponemos de una lista de constantes en *keyEvent* para cada tipo de pulsación y para gestionar cada evento.

Además de estar atento a las pulsaciones del teclado sería muy práctico que cada vez que se pulsara la tecla *enter* o se pulsará el botón *es primo?* el teclado desapareciera de la pantalla dejando todo para la visualización y si el lector fue realizando el ejemplo expuesto seguramente habrá comprobado que si no se escribe nada y se pulsa *enter* o el botón *es primo?* la aplicación se bloquea, para solucionar este problema y añadir la funcionalidad anterior se modificará el método *tarea* de la siguiente manera.


```

public void tarea(){
    InputMethodManager imm = (InputMethodManager) getSystemService(Context.INPUT_METHOD_SERVICE);
    imm.hideSoftInputFromWindow(entrada.getApplicationWindowToken(), 0);
    String aux=entrada.getText().toString();
    if(!aux.isEmpty()){
        int num=Integer.parseInt(aux);
        if(esPrimo(num))
            salida.setText("El número "+ num + " es primo");
        else
            salida.setText("El número "+ num + " NO es primo");
    }
}

```

Fig. 54. Modificación del método tarea.

La primeras dos líneas de código sirven para ocultar el teclado cada vez que se pulsa la tecla *intro* o el botón, lo primero que hacemos es tener acceso a la gestión del teclado virtual y lo segundo es ocultarlo cada vez que se haya pulsado el *EditText* *entrada* que es el que provoca que se muestre en pantalla ya que si no fue tocado el teclado no aparecerá.

La forma de solucionar el bloqueo por no escribir nada es muy sencilla, simplemente comprobamos que la cadena de texto leída no se encuentra vacía antes de hacer la conversación a entero para comprobar el número.

Finalmente se cambia en el diseño xml para que la alineación sea al centro del dispositivo y así provocar que sea el dispositivo que sea y la orientación de este todos los elementos se vean igual.

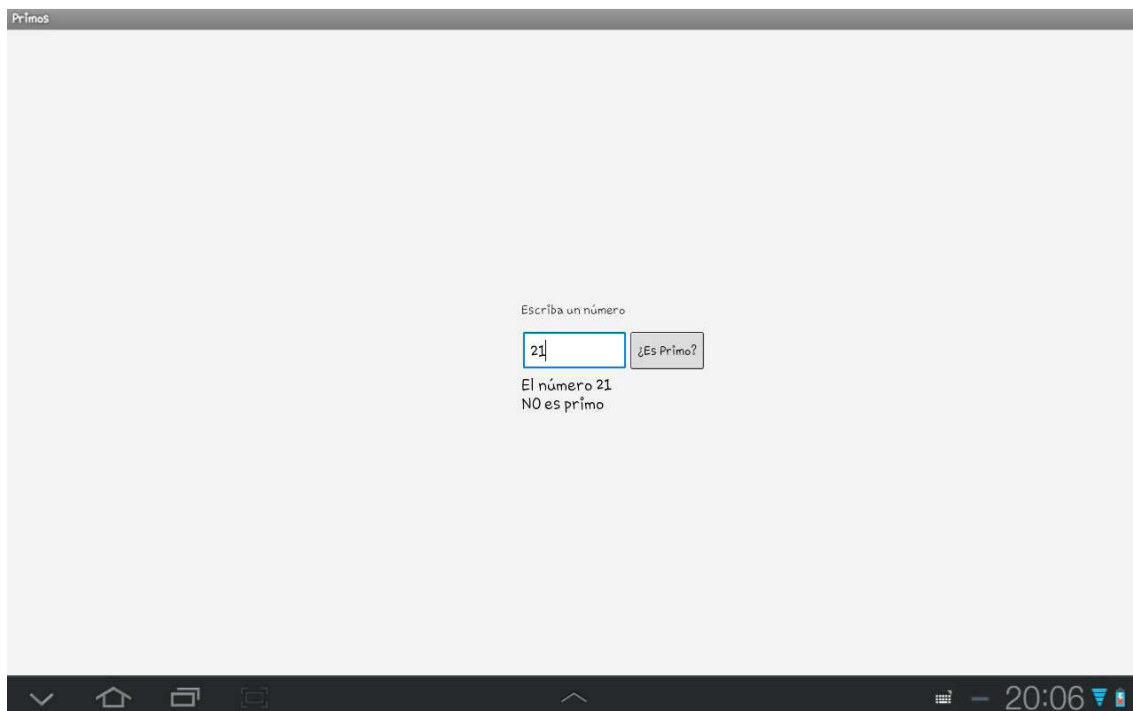


Fig. 55. Aplicación final de números primos

Como todas las aplicaciones de este proyecto el código se encuentra en el Anexo X, separado los .java y los .xml de cada aplicación.

2.1.7 La clase View y sus Subclases.

View representa el bloque de construcción básico para los componentes de interfaz de usuario, como se pudo observar en el ejemplo descrito y como se verá a lo largo de esta memoria. Una view ocupa una área rectangular en la pantalla del dispositivo siendo el responsable de su propio diseño y de la gestión de eventos (como el ejemplo del botón). Además disponemos de la subclase ViewGroup cuya función no es la representación de un elemento en pantalla sin la agrupación de elementos View o otros elementos ViewGroup comportándose como el patrón de diseño Composite.

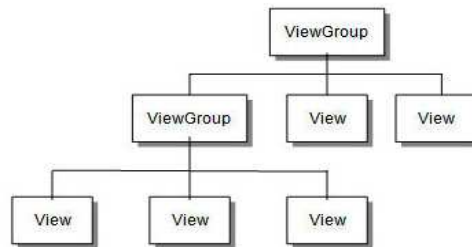


Fig. 56 ViewGroup

Cada activity tiene un ViewGroup del cual cuelgan todos los demás y es elemento base que asociamos nada más iniciar en el método *onCreate(Bundle bundle)* con la llamada *setContentview(R.layout.activity_main)*, no quiere decir que en Android todo depende de las view en la elaboración del diseño, ya que por ejemplo, se dispone de la clase Canvas de Java, pero son una herramienta básica para la gran mayoría de aplicaciones, se dispone de una gran cantidad de subclases de view como se puede ver en el siguiente diagrama gran parte de ellas y sus relaciones de herencia .

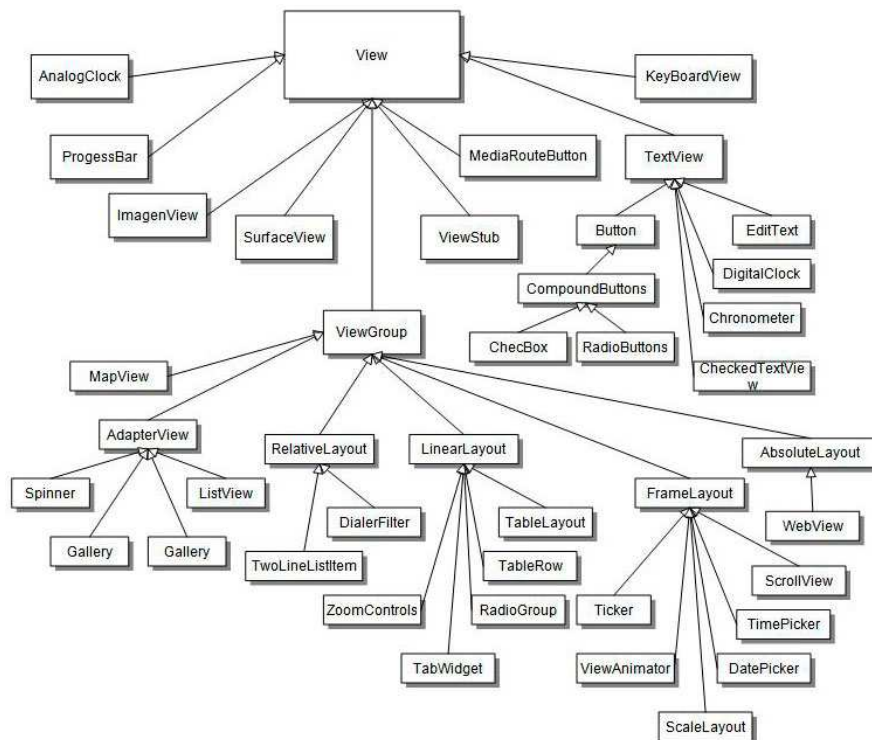


Fig. 57. View y sus subclases

Además del diseño y herramientas que se nos ofrece desde la vista de XML, mediante los métodos que se ofrecen la clase View y sus subclases se puede configurar el diseño, enfoque, eventos, colores y básicamente todo lo que se necesite. Para conocer los métodos que disponemos para cada view y que se puede hacer con ellos lo mejor es visitar la página <http://developer.android.com/intl/es/reference/android/view/View.html>.

2.2 2ª Aplicación. Servicio.

En esta segunda aplicación se presentará uno de las clases más importantes junto a las activity, los services. El diseño de la aplicación será simple, se tendrá una única activity que mostrará los mensajes por pantalla que le llegarán desde un service que ejecutara una tarea (hilo) de forma cíclica, el resultado es que se puede ver en la imagen.



Fig 58. Imagen de la aplicación Servicio.

Como se observa la imagen muestra dos mensajes, uno el momento de creación del servicio (Si ya esta arrancado) y otro que se actualizara cada 5 segundos siempre y cuando el servicio este iniciado y se haya pulsado el botón *Iniciar servicio*, y para detener el service se deberá pulsar el botón *Parar Servicio* porque como se explicara más adelante el hecho de salir de la activity no supondrá detener el service.

La creación de la aplicación seguirá un proceso idéntico al de la primera aplicación *Números primos*, con una única activity y un solo layout, que contendrá los elementos que se pueden observar en la figura 57 (El texto *Conectado* no está incluido en el layout, es un elemento Toast que se explicará a continuación).

2.2.1 Toast

Los elementos de clase Toast sirven mayoritariamente para lanzar mensajes emergentes desde cualquier parte del código de la aplicación. Aunque se pueden construir objetos Toast, esta clase nos ofrece un par de métodos estáticos que pueden ser utilizados para poder mostrar dichos mensajes. El método principal es el siguiente *public static Toast makeText (Context context, CharSequence text, int duration)* con él se indica en donde se ejecutará, que mensaje se mostrará y durante cuánto tiempo mediante dos variables definidas una para mostrarla durante un periodo breve de tiempo *Toast.LENGTH_SHORT* y la otra para un periodo más largo *Toast.LENGTH_LONG*. Un ejemplo de uso sería el siguiente.

```
Toast.makeText(this, "Desconectado", Toast.LENGTH_SHORT).show();
```

Fig. 59. Ejemplo de un Toast

Además como se observa en la imagen el método está acompañado de otro método, *.show()*, que es el encargado de hacer visible el Toast durante la duración indicada.

Existen otras herramientas para mostrar mensajes o ventanas emergentes más elaboradas que serán estudiadas en próximos capítulos.

2.2.2 La clase Service.

A diferencia de la clase *activity* un *service* no dispone de interfaz de usuario, funciona en un segundo plano en el cual realiza una determinada tarea, hay dos ideas muy comunes cuando se habla de *Services* la primera es pensar que son un proceso independiente al de la aplicación y la otra es que tienen un hilo independiente, esto no es así, los *services* forman parte del proceso principal de la aplicación, y lógicamente, no disponen de hilo propio, sino que dependen del hilo *main* de la aplicación.

Por lo tanto si se desea que haga algo de forma independiente a la aplicación deberemos iniciar un hilo dentro de esta, que es una de las formas más comunes de realización aplicaciones multitarea.

2.2.2.1 El ciclo de vida de Service.

La clase de *service* tiene dos posibles ciclos de vida que dependerán de la implementación que se haga, en la siguiente imagen se pueden observar.

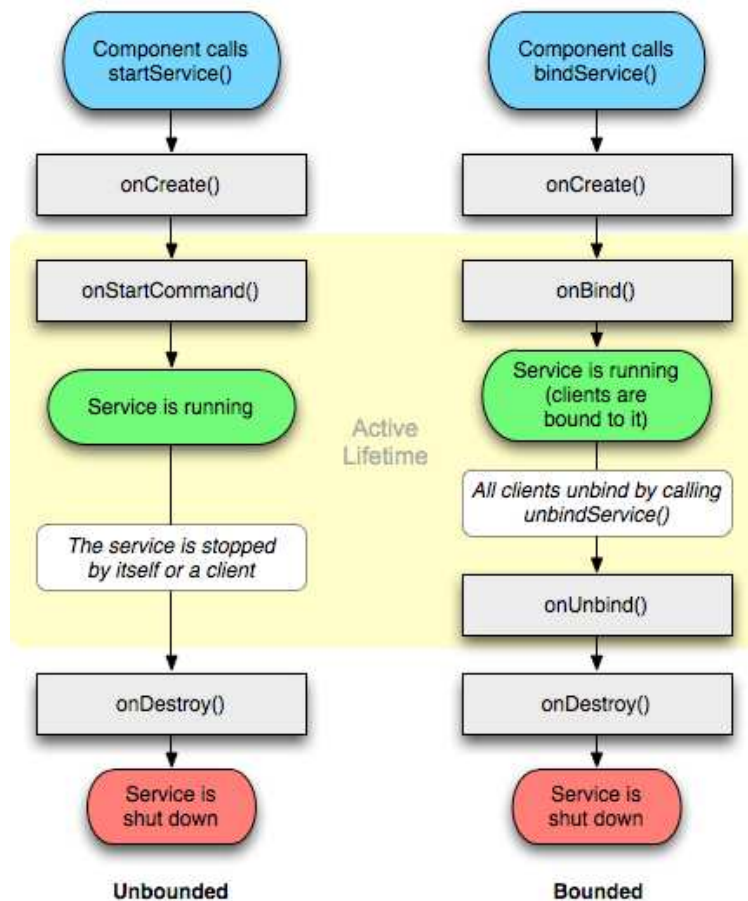


Fig. 60. Ciclos de vida de un service.

Como se puede observar las dos opciones disponen de un método onCreate que al igual que en las activities se utilizara para inicializar las variables y elementos necesarios.

El ciclo que se ve a la izquierda de la imagen recibe el nombre de no enlazado (*Unbounded*) y el de la izquierda de enlazado (*Bounded*).

En el ciclo no enlazado un service es llamado con *Context.startService()*, si el servicio no está creado éste se creará (se ejecutara el método onCreate()) y seguidamente se ejecutará el método *onStartCommand(Intent, int, int)* donde el cliente deberá especificar los elementos de entrada. El service puede detenerse desde el cliente mediante *Context.stopService()* o desde el mismo mediante *stopSelf()*. Hay que tener en cuenta que múltiples llamadas *Context.startService()* no hacen nido (aunque si darán lugar a múltiples llamadas *onStartCommand()*), así que no importa cuántas veces se ha iniciado el servicio se detendrá una vez se llamen a *Context.stopService()* o *stopSelf()* desde el mismo, sin embargo los servicios pueden utilizar el método *stopSelf(int)* para garantizar que el servicio no se detiene hasta que todos los intentos hayan sido procesados.

El ciclo de vida enlazada, los clientes pueden obtener una conexión persistentes con el service mediante el método *Context.bindService()*, el cual también crea el service

sino lo estuviera, pero no llama al *onStartCommand(Intent, int, int)*. El cliente recibirá el objeto *Ibinder* desde el método del servicio *onBind(Intent)* de esta manera puede intercambiar mensajes entre activity y services la aplicación Servicios. Hay que tener en cuenta que el service puede seguir funcionando aunque la activity haya finalizado porque no depende de una activity sino de la aplicación como ya se comento. Para poder realizar esto último se deberá añadir *onStartCommand* con un determinado valor de retorno, en esta aplicación veremos uno de ellos el resto se pueden consultar en la página de referencia. Para finalizar la conexión con el service se dispone del método *Context.unbindService(ServiceConnection)*.

El sistema operativo tratará de mantener con vida el proceso service siempre y cuando la aplicación con la que se iniciará este funcionando o aquellas que se vincularon con el sigan funcionando, la posibilidad de que un service sea eliminado de memoria sigue el siguiente orden de menor a mayor:

- Si está ejecutando código, es decir, se encuentra en los métodos *onCreate()*, *onStartCommand()* o *onDestroy()*, el service se considerará que esta interactuando como un activity que está en primer plano y no podrá ser eliminado.
- Si el proceso no se encuentra en uno de estos métodos pero está realizando alguna tarea en segundo plano, esto será lo normal en la aplicación servicio, el sistema operativo considerará que estará en una de las primeras posiciones de la pila y salvo casos de extrema necesidad de recursos no será eliminada de memoria.
- Si hay clientes vinculados al servicio, entonces el proceso que aloja el servicio será igual de importante en la pila que el cliente que se vinculo a él.
- Además se dispone del método *startForeground(int, Notification)* el cual indica al sistema operativo que el servicio está en primer plano y que el usuario es consciente de esto y por tanto no es candidato a ser eliminado de memoria.

2.2.2.2 Elementos del Service de la aplicación Servicios.

Aunque se han visto dos ciclos posibles de vida es muy común utilizar elementos de ambos en un mismo service, en el caso de la aplicación Servicios su service utilizará el método *onStartCommand()*, a continuación se presentará el ciclo de vida de dicho service y la función que se destino a cada parte de esté.

1. *Atributos*. Se dispone de dos elementos messenger Actividad y Servicio, los cuales se explicará en el siguiente punto su función; una variable boolean para indicar si el service está en marcha; un String; un Timer, estos objetos permiten ejecutar una determinada tarea en un hilo independiente repitiéndose pasado un tiempo dato, esté ultimo será clave en está aplicación, y además una serie de valores enteros constantes.
2. *onCreate*. Como se comento esté método será ejecutado cuando alguien llame al método *startService(Intent)* y el servicio no estaba creado, en este se define el

elemento messenger Servicio y se indica que el servicio está funcionando con la variable booleana.

```
public void onCreate() {
    super.onCreate();

    servicio = new Messenger(new Handler(this));

    isRunning = true;
}
```

Fig. 61. Método *onCreate()* del service de la aplicación Servicio.

3. *onStartCommand*. Cada vez que se llama mediante *startService(Intent)* será atendida en este método se podrá utilizar la información del objeto Intent los cuales se explicarán más adelante además el valor de retorno indicaremos el comportamiento del service en nuestro caso utilizamos la constante definida *START_NOT_STICKY*, con la que indicamos que el service habrá que cerrarlo de forma específica, es decir que aunque la activity que lo invoco se cerrase, si este no ha sido cerrado seguirá estando activo.

```
public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);

    fecha = intent.getStringExtra("fecha");

    return START_NOT_STICKY;
}
```

Fig. 62 Método *onStartCommand()* del service de la aplicación Servicio.

4. *onBind*. Aquí se devolverá el objeto el Binder del Messenger del Servicio, con el que el invocador del método *bindService()* pueda realizar la comunicación entre ambos procesos, en este momento la activity y el service estarían enlazados.

```
public IBinder onBind(Intent intent) {
    return servicio.getBinder();
}
```

Fig. 63 Método *onBind()* del service de la aplicación Servicio.

5. *handleMessage*. Por ahora solo indicaremos que es el encargado de gestionar los *message* que llegan al servicio, pero será uno de los elementos que se analizaran en el siguiente punto.
6. La clase *Tarea* que es heredera de la clase *TimerTask*, en la que se implementara un método *run* encargado de obtener la hora actual y pasarla como cadena de texto a la activity si se está conectada a ella o al propio service sino lo estuviera. La forma de construir la cadena de texto con la hora actual es la siguiente.

```
String time = new SimpleDateFormat("dd/MM/yyyy kk:mm:ss").format(new Date());
```

Fig. 64. Crear una cadena de texto con la hora actual.

2.2.3 Messenger, Handler, HandlerMessage y Message.

Estas herramientas nos permitirán intercambiar mensajes entre procesos de forma sencilla, lo primero que se tiene que hacer es que la activity o services correspondiente implemente Callback.

```
public class Servicio extends Service implements Callback{
```

Fig. 65. Service que implementa Callback.

Esto nos obligará a implementar el método `handleMessage(Message)` que gestionará todos los mensajes que lleguen a nuestro Messenger desde nuestro manejador.

```
public boolean handleMessage(Message msg) {
    switch (msg.what) {
        /*
         * Al responder al CONECTAR, el servicio
         * enviará la fecha en la que se creó.
         */
        case Servicio.CONECTAR:
            String fechaCreacion = (String) msg.obj;
            creacion.setText("Hora creación del Servicio: " + fechaCreacion);
            break;

        /*
         * Cuando nos llegue un MENSAJE, el servicio nos
         * estará enviando la hora actual.
         */
        case Servicio.MENSAJE:
            String fecha = (String) msg.obj;
            actual.setText("Hora actual del Servicio: " + fecha);
            break;
    }
    return true;
}

public boolean handleMessage(Message msg) {
    /*
     * En el campo "what" se especifica que tipo
     * de mensaje estamos enviando, así podremos
     * identificar lo que nos llegan y hacer el
     * casting de obj a lo que corresponda.
     */
    switch (msg.what) {
        case CONECTAR:
            /*
             * Si estamos conectando, en replyTo se tendrá el
             * Messenger de la actividad que lo he envío, así
             * que se guardará para cuando se quiera enviarle
             * algo.
             */
            activity = msg.replyTo;

            //Contextación
            Message m = new Message();
            m.what = CONECTAR; //Se especifica el tipo de mensaje
            m.obj = fecha; //Se Adjunta el objeto que queremos enviar
            try {
                activity.send(m); //Enviar
            } catch (RemoteException e) {}
        }
    }
}
```

Fig. 66. Implementación de `handleMessage(Message)` en la Activity y en el Service.

Los elementos básicos de los mensajes (objetos Message) que se intercambian entre los procesos y que podemos observar en las imagen superior son los siguientes:

- *what*, es una variable de tipo entero, identifica el tipo de mensaje no todo los mensajes se deben tratar igual como se puede ver en la imagen.
- *replyTo*, es un atributo de tipo *Messenger* opcional, el cual se utiliza para indicar a quien deben enviarse los mensajes de contestación, que no siempre tendrá porque ser el mismo que los envió.
- *obj*, es un elemento *Object* aquí podemos encapsular los datos que se desean mandar como por ejemplo se hace en la parte izquierda inferior de la imagen donde se encapsulan un String para su envío.
- *arg1* y *arg2*, son dos variables enteras utilizadas como *what* pero totalmente opcionales. Su utilidad, por norma general, es la de identificar aun más el tipo de mensaje.

Una vez que se conocen los mensajes que se intercambian entre los procesos podemos explicar el patrón de funcionamiento que hay detrás de este proceso.

Si observamos la activity podemos identificar en los atributos dos elementos Messenger, y en onCreate el constructor de uno de ellos como vemos en la imagen.


```

private Messenger activity;
private Messenger servicio;

//Creamos el Messenger para el activity
activity = new Messenger(new Handler(this));

```

Fig. 67. Creación de Messenger en la activity

Lo que estamos indicando es que tenemos un elemento Messenger que utilizara un manejador que será el propio contexto de la activity, es decir, que los mensajes que se manden a este proceso activity serán gestionados por este elemento.

Por otro lado hemos indicado que nuestra clase activity será la encargada de gestionar una comunicación con un service, para ello implementará *ServiceConnection* como se observa en la imagen.

```

public class Actividad extends Activity implements ServiceConnection, Callback {

```

Fig. 68. Declaración de la activity Actividad

Esta implementación nos obliga a utilizar los métodos *onServiceConnect()* y *onServiceDisconnect()*, este último se ejecutara después de que se llama al método *unBindService()* y en nuestro caso solo mostrará un Toast por pantalla. El primero de ellos se ejecuta en el momento en que se consiguió enlazar con la actividad donde el tipo de entrada será el enviado en el método *onBind* del service, es decir, un IBinder el cual contendrá la dirección del Messenger por el cual se comunicará la activity con el service.

```

public void onServiceConnected(ComponentName name, IBinder service) {
    servicio = new Messenger(service);

    Message msg = new Message();
    msg.what = Servicio.CONECTAR;
    msg.replyTo = activity;

    try {
        servicio.send(msg);
    } catch (RemoteException e) {
    }
}

```

Fig. 69. *onServiceConnected* de la activity Actividad.

El método superior será el encargado de mandar el primer message al service indicándole al receptor en el campo *replyTo* que deberá contestar al messenger de la propia activity. Más adelante se entrará en el proceso que se recorrerá para llegar a ejecutar este método.

2.2.4 Intent

Es una de las herramientas más importantes que se ofrece a los desarrolladores de Android. Según la definición dada en la página de referencia de desarrollo de Android un intent es una descripción abstracta de una operación a ser realizada, estos pueden ser utilizados en diferentes ámbitos para empezar una nueva activity o service, en el archivo de AndroidManifest.xml, enlazarse a un service como se verá en la aplicación Servicio entre otras operaciones.

Las clases Intent son la red de comunicación de las aplicaciones Android. En muchos aspectos, la arquitectura de Android es similar a una arquitectura Orientada a Servicios, ya que cada actividad realiza un tipo de invocación intent para ejecutar una operación, sin saber quién será el receptor.

Un intent proporciona de manera sencilla la realización de enlaces y comunicación entre aplicaciones en tiempo de ejecución. Su uso más importante es la puesta en marcha de actividades, en este proceso el intent se puede ver como punto de nexos entre ellas. Se trata básicamente de una estructura de datos pasiva que contiene una descripción abstracta de una tarea a realizar, sin embargo, en esta segunda aplicación no se entrará en toda la funcionalidad de la clase intent, aunque en la tercera aplicación se verán ejemplos más profundos sobre esta.

2.2.4.1 Estructura de un intent

Un intent se compone de dos grupos de elementos, las acciones y los datos, la forma de entender esto será con un sencillo ejemplo, si se desea realizar una aplicación que regule el volumen con el que suenan las melodías de una llamada entrante la acción contenida en el intent sería el de modificar el volumen, y los datos el nivel a que quisiéramos subirlo, este sería mandando a la aplicación que gestiona el sistema de audio de Android que deberá estar preparada para atender este intent y además resolver esta acción no solo en su código java, sino que esto deberá estar indicado en su AndroidManifest.xml.

Además de estos elementos básicos hay otros atributos secundarios que pueden ser incluidos en un intent.

- *Category*, da una información adicional de la acción a ejecutar.
- *Type*, especifica un tipo explícito (un tipo MIME) de los datos del intent. Normalmente el tipo se deduce a partir de los propios datos. Al establecer este atributo se desactiva la evaluación automática del tipo de dato explícito.
- *Component*, especifica el nombre de una clase componente que será usado para el propósito del intent.
- *Extra*, este es un *Bundle* (paquete) con información adicional, estos se utilizan para facilitar una información adicional para el componente. Por ejemplo si se desea realizar la acción de mandar un e-mail, se podrá incluir elementos de este como el cuerpo, asunto, adjuntos...

En el caso de la aplicación servicios el uso de los intent será para enlazar la activity y el service además de iniciar este último. Sin embargo, en siguientes aplicaciones se extenderá el uso de estos y de otra de las herramientas importantes que son los Intent Filters que se pueden ver en el archivo de AndroidManifest.xml

2.2.5 Funcionamiento de la Aplicación Servicio.

Ya se conocen los elementos de la clase service, las herramientas de comunicación para poder mandar mensajes sencillos entre procesos y los intent desde un punto de vista general. A lo largo de este punto se explicara desde el punto de vista del usuario el funcionamiento de la aplicación acompañando esta vista del código que hay tras ella.

Cuando se inicia la aplicación, se crea únicamente la activity con la vista definida en su correspondiente layout que podemos observar en la siguiente imagen acompañada del método onCreate donde se definen las primeras funcionalidades de la aplicación y los métodos que utilizan los botones.

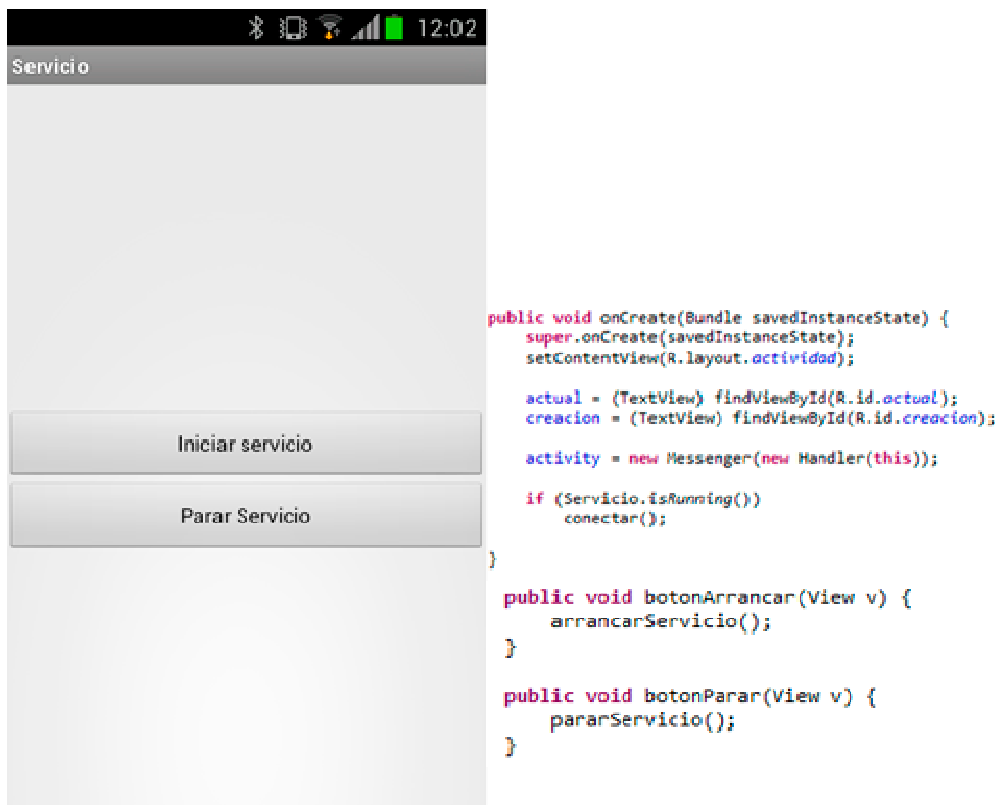


Fig. 70. Primera vista de la aplicación servicios y método onCreate de su activity

Inicialmente se dispone de dos botones, los cuales son atendidos sus eventos en los correspondientes métodos que se ven en la parte inferior de la imagen, por otro lado podemos ver que se relacionan dos *TextView* a través de su id, estos están situados en la parte superior de los botones el de nombre actual y de nombre creación en la parte inferior de estos. Además una vez hecho se construye el messenger activity, se comprueba si el service está corriendo en caso de ser así se podrá conectar directamente

a él, como veremos más adelante, ahora se supondrá que se está ejecutando la aplicación por primera vez.

Lo siguiente que se estudiara será el método *arrancarServicio()* que se desencadena al ser pulsado el botón *Iniciar servicio* y el método *conectar()*.

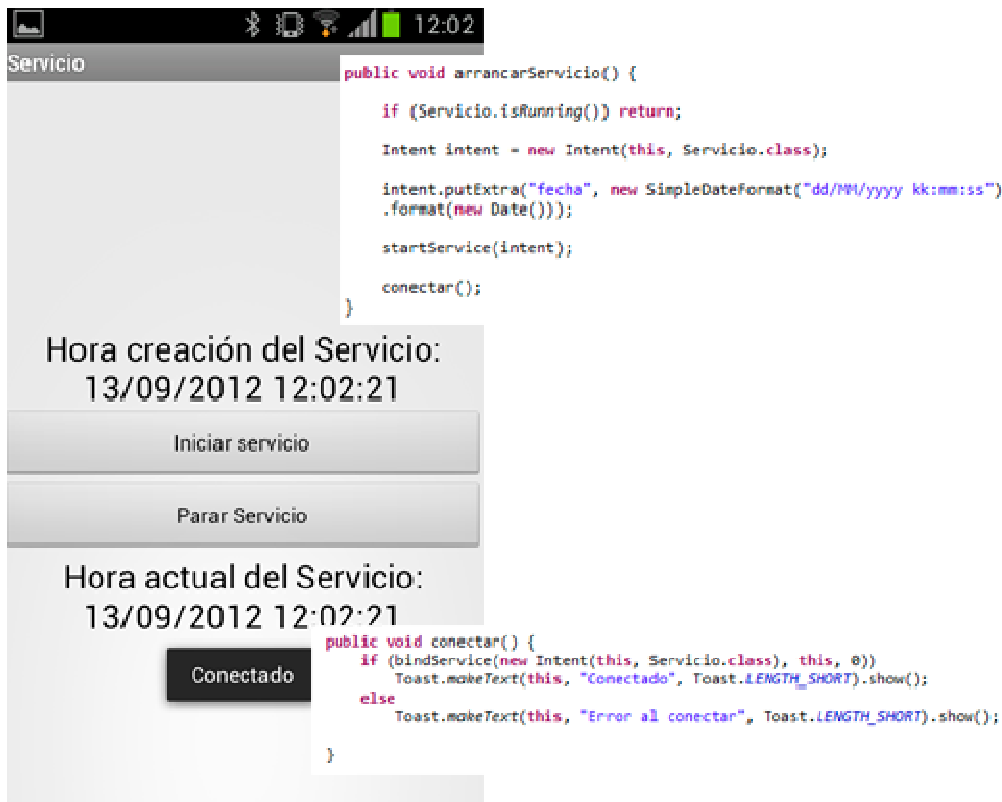


Fig. 71. Aplicación servicios tras pulsarse *Iniciar servicio*

Lo primero que hace el método *arrancarServicio()* es comprobar si el servicio está funcionando o no, si lo está no se permite volver a ponerlo en marcha, lo siguiente será crear un nuevo intent, usando uno de sus constructores donde se indica quien va ser invocador del servicio y la clase que se quiere crear, además se añade la fecha actual, para ello se indica el nombre de la variable y por otro lado su valor. Una vez creado se pone en marcha el servicio al ejecutar el método con *startService(Intent)*, siendo el Intent el recién creado.

En ese momento, en el servidor se pone en marcha el service se iniciara su método *onCreate* y seguidamente se ejecuta el método *onStartCommand(Intent intent, int flags, int startId)* que se puede ver en la figura 61 en ella se puede ver como se llama al método *getStringExtra(String)* el cual contiene el mismo nombre que le dimos a la cadena que se añadió al intent y que devuelve el valor de la variable, en caso de poner un nombre que no se añadiera al intent esta llamada devolvería null, y como ya se comento, se indica que el servicio estará en marcha hasta que se cierre específicamente.

Seguidamente a esto se lanza el método *conectar*, su función será enlazar con el servicio siempre y cuando este lo permita, mediante la llamada al método

bindService(Intent, ServiceConnection, Int) el cual devuelve un boolean que nos indicará si se consiguió o no la conexión, con esta información se mostrará un mensaje Toast en pantalla indicando si se logro o hubo un fallo en la conexión. Si toda va bien, en ese momento se ejecutará el método *onBind* que como se ve en la figura 62, el cual devolverá un elemento *IBinder* que contendrá la dirección del messenger del servidor, dicho valor de retorno será atendido en el método *onServiceConnect(Component name, IBinder service)* como se puede observar en la imagen inferior.

```
public void onServiceConnected(ComponentName name, IBinder service) {
    servicio = new Messenger(service);

    Message msg = new Message();
    msg.what = Servicio.CONECTAR;
    msg.replyTo = activity;
    try {
        servicio.send(msg);
    } catch (RemoteException e) {
    }
}
```

Fig. 72. *onServiceConnected* de la activity de la aplicación Servicio.

Alcanzado este punto de la ejecución el proceso activity consiguió acceso al messenger del proceso del service y por tanto se puede iniciar una comunicación entre ambos, para que el service pueda comunicarse con la activity este deberá indicar su messenger al service para ello utilizara el campo *replyTo* del mensaje (*Message*) para indicar donde se desea recibir la respuesta. Además se indica que el mensaje que es de tipo *Servicio.CONECTAR* en el campo *what*, todo esto será enviado desde la activity al service usando su messenger el método *send(Message)* como se ve en la imagen, siendo este un proceso que puede incurrir en el lanzamiento de excepciones siendo necesarias tratarlas en el propio método.

Tras esto provocará la llegada de un *message* al service que como ya se indico será atendido por su método *HandlerMessage*, el cual se puede ver en la siguiente imagen.

```
public boolean handleMessage(Message msg) {
    switch (msg.what) {
        case CONECTAR:
            activity = msg.replyTo;
            Message m = new Message();
            m.what = CONECTAR;
            m.obj = fecha;
            try {
                activity.send(m);
            } catch (RemoteException e) {
            }
            if(tarea != null){
                tarea.cancel();
            }
            tarea = new Timer();
            tarea.schedule(new Tarea(), 0, 5*1000);
            break;

        case DESCONECTAR:
            activity = null;
            break;

        case MENSAJE:
            String texto = (String) msg.obj;
            Toast.makeText(this, texto, Toast.LENGTH_S
            break;
    }
    return true;
}
```

Fig. 73. Método *handleMessage* del service de la aplicación servicios

El método está preparado para recibir tres tipos de mensajes en este caso nos centraremos en el primero de ellos y el resto se irán resolviendo según se avance en el funcionamiento de la aplicación. El mensaje recibido desde la activity fue de tipo *CONECTAR*, lo primero será obtener el destinatario de los mensajes del service y se devuelve a la activity el mensaje con la fecha que recibió en el método *onStartCommand*, seguidamente se explicará que continua haciendo esta zona del método pero primero se verá como es tratado este mensaje en la activity.

```
public boolean handleMessage(Message msg) {  
  
    switch (msg.what) {  
        case Servicio.CONECTAR:  
            String fechaCreacion = (String) msg.obj;  
            creacion.setText("Hora creación del Servicio: " + fechaCreacion);  
            break;  
  
        case Servicio.MENSAJE:  
            String fecha = (String) msg.obj;  
            actual.setText("Hora actual del Servicio: " + fecha);  
            break;  
    }  
    return true;  
}
```

Fig. 74. Método *handleMessage* de la activity de la aplicación servicios

El método *handleMessage* de la activity es más sencillo que el del service, este solo está preparado para recibir dos tipos de mensajes *CONECTAR* y *MENSAJE* en el primero, que es el que recibimos, se carga el String contenido en el *obj* del mensaje en el *TextView* *creación* y en el segundo caso igual pero en el *TextView* *actual*. Por lo tanto la activity se queda esperando a que pasen tres posibles sucesos, uno que llegue un nuevo *message*, otro que se pulse el botón *Parar Servicio* y por último que se salga de la activity, que no es lo mismo que finalizar la aplicación.

Volviendo al servicio, se observa que comprueba si existe algún elemento tarea y de ser así es eliminado, seguidamente se crea un nuevo elemento *tarea* como un nuevo tipo *Timer* el cual es capaz de ejecutar un *TimerTask* y hacer que se repita su ejecución pasado un intervalo de tiempo definido, para poder realizar esto hace uso del método *schedule(TimerTask task, long delay, long period)* como se puede observar en la parte superior derecha de la figura 71.

El elemento que se introduce en el *TimerTask* es un elemento hijo de esta clase definido dentro del service ya que utilizara variables de este como se ve a continuación.

```

public class Tarea extends TimerTask {
    public void run() {
        try {
            String time = new SimpleDateFormat("dd/MM/yyyy kk:mm:ss").format(new Date());
            Message m = new Message();
            m.what = MENSAJE;
            m.obj = time;
            //Se comprueba si se puede mandar a la activity
            if (activity != null)
                activity.send(m);
            else
                servicio.send(m);
        } catch (RemoteException e) {
        }
    }
}

```

Fig. 75. Clase Tarea definida en la clase service

La *tarea* que se ejecutara cada 5000 milisegundos será mandar un string, con la fecha en ese momento, a la activity o al service como ya se comento en los elementos de este último. Por lo tanto, ya se tiene el primero de los posibles sucesos que pueden ocurrir en la activity que es que llegue un mensaje de tipo *MENSAJE* el cual como se vio se mostrara en el correspondiente *TextView*, y mientras que no se haga nada en la activity todo esto se estará repitiendo indefinidamente.

Ahora se supondrá que no se ha pulsado el botón parar y se pulsa el botón volver atrás del sistema que provocará en este caso que se ejecute el método *onDestroy* de la activity.

```

public void onDestroy() {
    desconectar();
    super.onDestroy();
}

public void onServiceDisconnected(ComponentName name) {
    Toast.makeText(this, "Desconectado", Toast.LENGTH_SHORT);
}

public void desconectar() {
    if(!Servicio.isRunning()){return;}

    Message msg = new Message();
    msg.what = Servicio.DESCONECTAR;

    try {
        servicio.send(msg);
    } catch (RemoteException e) {
    }

    if(Servicio.isRunning()){
        unbindService(this);
    }
}

```

Fig. 76. Métodos *onDestro*, *desconectar* y *onServiceDisconnected* de la activity.

El método *onDestroy()* es también muy sencillo, simplemente se ejecutará el método *desconectar()*, este método será el encargado de mandar un mensaje al service tipo *DESCONECTAR* y desvincularse de este, en ese momento se ejecutará el método *onServiceDisconnected (ComponentName name)* que mostrara un Toast como se puede observar, finalmente el método *onDestroy* se encargará seguir el proceso de eliminar la activity, ahora se estudiara como afecta el mensaje mandado en el service.

Como se observa en la figura 71, al llegar un mensaje de tipo *DESCONECTAR* simplemente se indica que no hay activity, lo cual provoca que los mensajes que son enviados desde la tarea a la activity sean dirigidos al propio service como se ve en la

figura, que cuando le lleguen estos mensajes los mostrará en el contexto en el que este la aplicación como en el *desktop* que se observa en la imagen inferior.

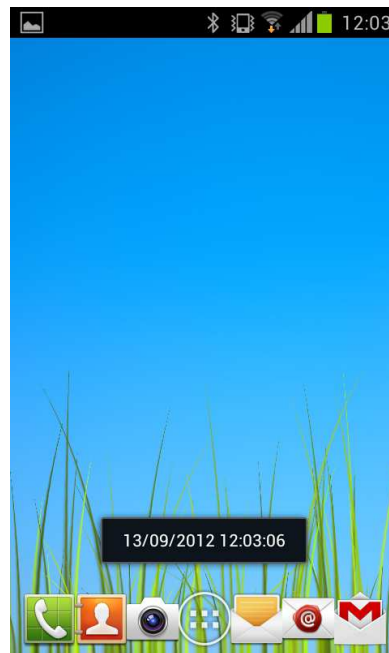


Fig. 77. Service en ejecución sin activity de la aplicación servicio

Para volver a recuperar el control del servicio, se iniciara nuevamente la aplicación que arrancara su única activity la cual comprueba si el service está arrancado y al estarlo la activity ejecutara directamente el método conectar como ya se vio anteriormente, por lo tanto solo falta ver el desencadenamiento del botón *Parar Servicio*.

```

public void pararServicio() {
    if (!Servicio.isRunning()) return;
    desconectar();
    stopService(new Intent(this, Servicio.class));
}

public void onDestroy() {
    if (tarea != null)
        tarea.cancel();

    isRunning = false;
    super.onDestroy();
}

```



Fig. 78. Métodos *pararServicio()* de la activity y *onDestroy()* del service, junto a imagen de la aplicación servicios, con el desencadenamiento de haber pulsado el botón *Parar Servicio*.

Cuando se pulsa el botón lo primero que provoca es la ejecución del método *desconectar()* que ya estudiamos que hacía y que provocaba a continuación ejecuta el método *StopService(Intent)* el cual provocará la ejecución del método *onDestroy()* en el service y por tanto la eliminación de esto.

Una aplicación sencilla pero que si se observo con detenimiento las zonas de código en la gran mayoría de ellas se tiene en cuenta el estado del service mediante la variable *isRunning* y de la tarea, ya que de no hacerse así provocaría situaciones inesperada y no deseadas de la aplicación, llegando a provocar bloqueos del sistema operativos y en algunos casos más extremos el reinicio del dispositivo.

2.3 3ª Aplicación. Sensores y Bluetooth

En esta segunda aplicación se avanzara en el ciclo de vida de las activity; se estudiara el acceso y uso de los datos de sensores, en este caso será el acelerómetro; se presentará una de las herramientas de conexión disponible en la gran mayoría de equipos Android, Bluetooth; y una pequeña presentación de representación usando Canvas en Android. Además de otros pequeños detalles y conceptos.

Para la realización de los objetivos marcados la aplicación de partida será un ejemplo de los que se nos ofrece en las API de Android sobre el acelerómetro, en la cual introduciremos los cambios necesarios para conseguir abarcar los conocimientos que no cubre inicialmente.

Esta es la vista final de la aplicación en un dispositivo (Un Samsung Galaxy S2, con Android 4.0.3) en ella se puede ver quince bolas plateadas moverse por la pantalla, además por detrás estará trabajando el bluetooth mandando información sobre el sensor a una aplicación simétrica pero que utilizara la información recibida para la representación.



Fig. 79. AceltometroPlay.

2.3.1 Crear un proyecto a través de un ejemplo.

Para ello se debe pulsar **ctrl+N** o en **File→New→Other** en el cual se mostrará la siguiente ventana en el cual deberemos elegir la opción **Android Sample Project**.

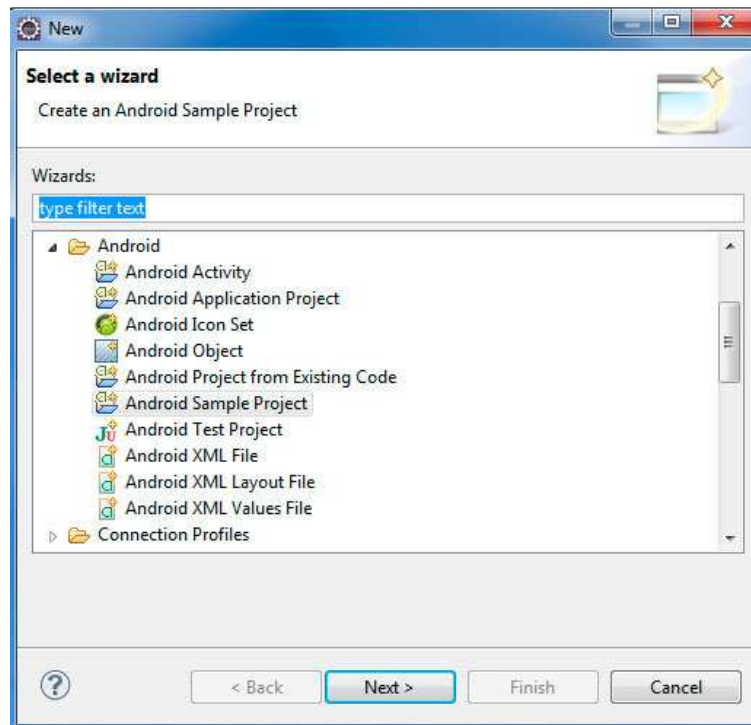


Fig. 80. Crear un nuevo proyecto a través de un ejemplo

En la siguiente ventana deberemos indicar de SDK de la se elegirá un ejemplo, en este caso se utilizara la versión 2.3.3, aunque en las versiones de *Android Open Source Project* se van añadiendo los ejemplos de las anteriores a cada una de las versiones nuevas, por ejemplo si accediéramos a los ejemplos de la versión 4.0 en adelante se podrían ver todos los ejemplos anteriores más los nuevos de NFC propios de esta versión.

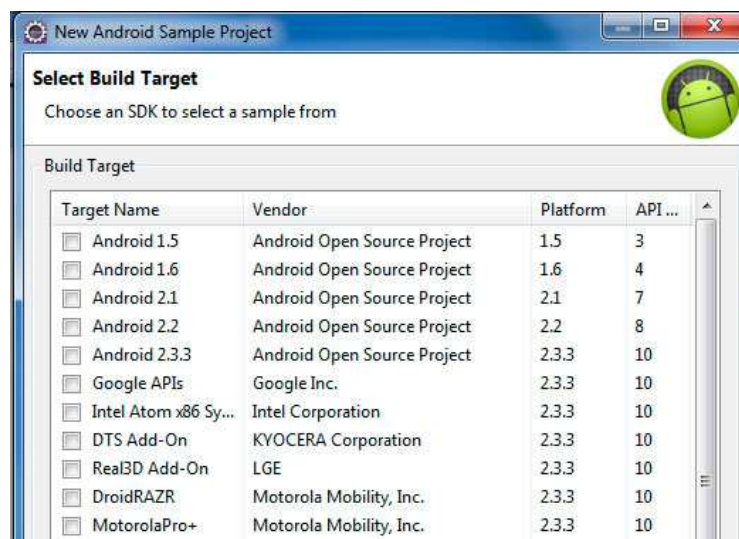


Fig. 81. Select Build Target

Después de seleccionar la versión, tendremos que indicar que ejemplo se quiere añadir, en nuestro caso el primero de ellos *AccelerometerPlay*.

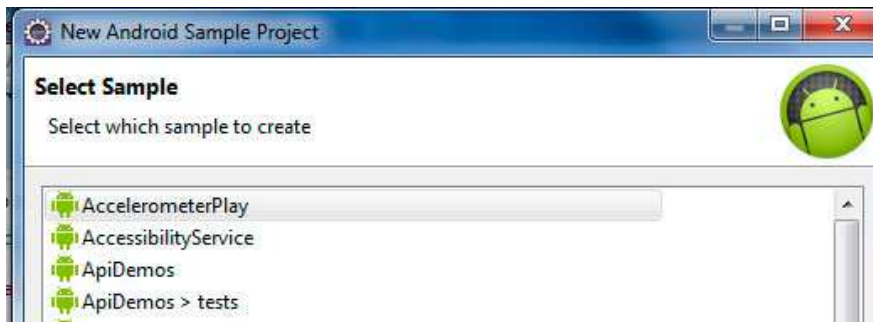


Fig. 82. Seleccionando un ejemplo

Una vez completado este último paso se tendrá en nuestro espacio de trabajo una versión completa del proyecto.

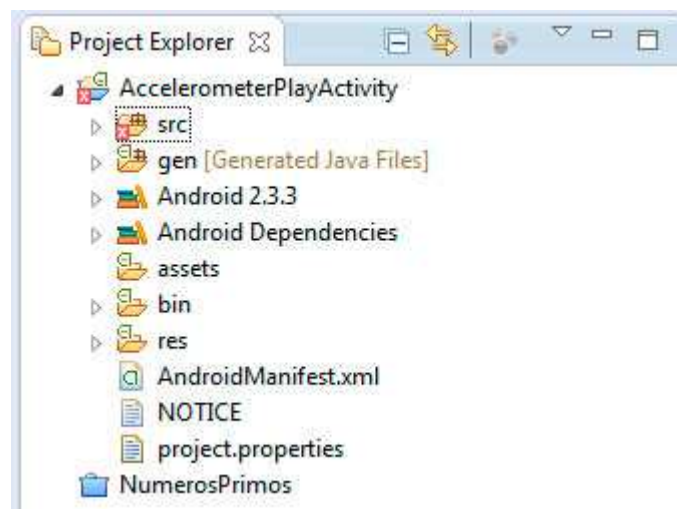


Fig. 83. Vista del espacio de trabajo.

Como se observa ya disponemos de un proyecto completo y también presenta algún tipo de error o conflicto, son, por norma general, pequeños problemas de texto de comentarios que no han sido marcados como tales, por lo tanto con las opciones de depuración ofrecidas se pueden solucionar estos problemas sin mayor importancia.

2.3.2 Funcionamiento del AccelerometerPlay

El primer paso será abrir la clase *AccelerometerPlayActivity.java* en él una vez solucionado los problemas ocultaremos la implementación de cada método o clase definida en él para tener una visión del conjunto que disponemos, para ello pulsaremos el símbolo "-" que aparece junto al enunciado de cada método.

```

public class AccelerometerPlayActivity extends Activity {

    private SimulationView mSimulationView;
    private SensorManager mSensorManager;
    private PowerManager mPowerManager;
    private WindowManager mWindowManager;
    private Display mDisplay;
    private WakeLock mWakeLock;

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

    protected void onResume() {}

    protected void onPause() {}

    class SimulationView extends View implements SensorEventListener {}
}

```

Fig. 84. AccelerometerPlayActivity.java

Si se hizo de forma correcta esta deberá ser la vista actual de la activity, lo primero que se observa es la definición de atributos disponemos de varios para la gestión de sensores, pantalla... y un elemento SimulationView.

Se dispone de un método *onCreate*, *onResume* y *onPause*, definidos en el ciclo de vida de una activity, además se ha creado dentro de la activity la clase *SimulationView* que hereda sus propiedades de *View* e implementa *SensorEventListener*. Igualmente si se extiende dicha clase se podrá observar que dentro de ella se han definido dos clases más, la clase partícula y la clase sistema de partículas (particle y particleSystem respectivamente).

El primer objetivo será conocer estos elementos y cómo actúan dentro de esta aplicación base y conocer qué función desempeña en el método *onCreate*.

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

    mPowerManager = (PowerManager) getSystemService(POWER_SERVICE);

    mWindowManager = (WindowManager) getSystemService(WINDOW_SERVICE);
    mDisplay = mWindowManager.getDefaultDisplay();

    mWakeLock = mPowerManager.newWakeLock(PowerManager.SCREEN_BRIGHT_WAKE_LOCK, getClass()
        .getName());

    mSimulationView = new SimulationView(this);
    setContentView(mSimulationView);
}

```

Fig. 85. Método *onCreate* de AccelerometerPlayActivity.

Cada uno de los elementos que se crean tienen un objetivo el cual se describe a continuación hay que tener en cuenta que el tipo es el mismo que el nombre pero sin *m*.

- *mSensorManager*, con la llamada que se hace se obtiene una instancia que nos permitirá acceder a los sensores del dispositivo.

- *mPowerManager*, con la invocación que se hace se obtiene una instancia del sistema encargado de detectar cambios como el bloqueo de pantalla o el encendido y apagado de esta.
- *mWindowsManager*, con la llamada que se hace se obtiene una instancia para poder controlar el estado de la ventana principal del dispositivo, su orientación y sus cambios entre otras tareas.
- *mDisplay*, se usa uno de los métodos que nos ofrece el anterior objeto, con el podremos detectar los cambios que se producen en la orientación del dispositivo y que se utilizarán más adelante en el uso del sensor.
- *mWakeLock*, la implementación de esta variable es a través del método *newWakeLock* del objeto *mPowerManager* para tener un control total del momento en el que se bloquee o desbloquee el dispositivo no solamente cuando se apague o encienda la pantalla de este.
- *mSimulationView*, es un objeto de la clase *SimulationView* que se puede observar en la figura 82, el cual será el encargado de la interfaz de usuario y de controlar el sensor del acelerómetro, por ese el elemento que se le pasa al método *setContentview* y no un layout como se realizó en las anteriores aplicaciones. En siguientes puntos se explicará cómo funciona y de que está compuesto.

Lo siguiente será conocer qué función desempeña el método *onResume()*, que como ya se comentó será el siguiente en ser ejecutado en esta actividad ya que no se implementó el método *onStart()*.

```
protected void onResume() {
    super.onResume();

    mWakeLock.acquire();

    mSimulationView.startSimulation();
}
```

Fig. 86. Método *onResume* de *AccelerometerPlayActivity*.

La llamada *mWakeLock.acquire()*, se ha asegurado que el dispositivo está desbloqueado antes de seguir haciendo cualquier otra tarea, una vez hecha esta comprobación llama al método *startSimulation()* de la simulación. El siguiente elemento a estudiar es el método *onPause* que se ejecutará cuando se bloquee el dispositivo o se pase a segundo plano tras pulsar el botón de menú del dispositivo.

```
protected void onPause() {
    super.onPause();

    mSimulationView.stopSimulation();

    mWakeLock.release();
}
```

Fig. 87. Método *onPause* de *AccelerometerPlayActivity*.

Lo primero que hace es llamar al método *stopSimulation()* de la simulación e indica que hasta que no haya una vuelta a la aplicación no se vuelva al método *onResume()* mediante el método .

Por último, falta por estudiar el elemento más importante de esta aplicación, la clase *SimulationView* que será la encargada de gestionar la interfaz de usuario y hacer uso del sensor.

```
class SimulationView extends View implements SensorEventListener {  
  
    private static final float sBallDiameter = 0.004f;  
    private static final float sBallDiameter2 = sBallDiameter * sBallDiameter;  
  
    private static final float sFriction = 0.1f;  
  
    private Sensor mAccelerometer;  
    private long mLastT;  
    private float mLastDeltaT;  
  
    private float mXDpi;  
    private float mYDpi;  
    private float mMetersToPixelsX;  
    private float mMetersToPixelsY;  
    private Bitmap mBitmap;  
    private Bitmap mWood;  
    private float mXOrigin;  
    private float mYOrigin;  
    private float mSensorX;  
    private float mSensorY;  
    private long mSensorTimeStamp;  
    private long mCpuTimeStamp;  
    private float mHorizontalBound;  
    private float mVerticalBound;  
    private final ParticleSystem mParticleSystem = new ParticleSystem();  
}
```

Fig. 88. Clase *SimulationView*

Lo primero ver que hereda las propiedades de un *View*, lo que nos obligara a reescribir el constructor de la clase y por otro lado implementa la interfaz *SensorEventListener* con el que se compromete a tener los métodos *onAccuracyChanged(Sensor sensor, int accuracy)* y *onSensorChanged (SensorEvent event)* que se explicará más adelante su función, además como se observa en la imagen tenemos gran cantidad de objetos que se utilizaran para la representación grafica y para calcular los cambios que se realizan en esta, de entre todas estas variables habrá que resaltar dos, la primera la variable *sensor mAccelerometer* que será la encargada de gestionar el sensor del acelerómetro y la segunda la variable *mParticleSystem* que se declara y construye en la misma declaración de atributos de la clase y serán la encargada de controlar las bolas que se moverán por la pantalla.

Para poder entender cómo funciona la representación grafica como conjunto se debe entender primero los elementos que componen esta, el primer elemento de estudio será la clase *Particle* que es la que define cada una de las bolas que se ven en la pantalla.

```

class Particle {
    private float mPosX;
    private float mPosY;
    private float mAccelX;
    private float mAccelY;
    private float mLastPosX;
    private float mLastPosY;
    private float mOneMinusFriction;

    Particle() {
        final float r = ((float) Math.random() - 0.5f) * 0.2f;
        mOneMinusFriction = 1.0f - sFriction + r;
    }

    public void computePhysics(float sx, float sy, float dT, float dTC) {

    }

    public void resolveCollisionWithBounds() {
        final float xmax = mHorizontalBound;
        final float ymax = mVerticalBound;
        final float x = mPosX;
        final float y = mPosY;
        if (x > xmax) {
            mPosX = xmax;
        } else if (x < -xmax) {
            mPosX = -xmax;
        }
        if (y > ymax) {
            mPosY = ymax;
        } else if (y < -ymax) {
            mPosY = -ymax;
        }
    }
}

```

Fig 89. Clase *particle*

Cada bola tiene una serie de variables que indican su posición actual, su aceleración respecto los ejes X e Y, su anterior posición y una fricción propia sobre la superficie virtual sobre la que se mueven, dicha fricción se calcula en su constructor, con todo estas variables y el constructor tenemos dos métodos uno de ellos que tenemos desplegado resuelve la colisión con los bordes de la pantalla de tal manera que las bolas siempre están contenidas en ella. El otro se puede observarlo a continuación.

```

public void computePhysics(float sx, float sy, float dT, float dTC) {
    final float m = 1000.0f;
    final float gx = -sx * m;
    final float gy = -sy * m;
    final float invm = 1.0f / m;
    final float ax = gx * invm;
    final float ay = gy * invm;
    final float dTdT = dT * dT;
    final float x = mPosX + mOneMinusFriction * dTC * (mPosX - mLastPosX) + mAccelX
        * dTdT;
    final float y = mPosY + mOneMinusFriction * dTC * (mPosY - mLastPosY) + mAccelY
        * dTdT;
    mLastPosX = mPosX;
    mLastPosY = mPosY;
    mPosX = x;
    mPosY = y;
    mAccelX = ax;
    mAccelY = ay;
}

```

Fig. 90. Método *computePhysics* de la clase *Particle*

Como se puede observar en él se calcula la posición en la que se mueve cada bola dado las correspondientes aceleraciones respecto los ejes y unos intervalos de tiempo.

El siguiente elemento de estudio es la clase *ParticleSystem* el cual se encarga de gestionar todas las *particle*. Los únicos atributos que dispone son una variable entera estática de valor quince y nombre *NUM_PARTICLES* y un array de elementos *Particle* del tamaño de la variable anterior.

```
class ParticleSystem {
    static final int NUM_PARTICLES = 15;
    private Particle mBalls[] = new Particle[NUM_PARTICLES];

    ParticleSystem() {
        for (int i = 0; i < mBalls.length; i++) {
            mBalls[i] = new Particle();
        }
    }
    private void updatePositions(float sx, float sy, long timestamp) {}
    public void update(float sx, float sy, long now) {}

    public int getParticleCount() {
        return mBalls.length;
    }

    public float getPosX(int i) {
        return mBalls[i].mPosX;
    }

    public float getPosY(int i) {
        return mBalls[i].mPosY;
    }
}
```

Fig. 91. Clase *ParticleSystem*.

Igualmente se disponen de un constructor el cual crea cada uno de los elementos del array y de cinco métodos tres de ellos muy sencillos que podemos ver desplegados en la imagen, que sirven para obtener información relativa al sistema de bolas o sobre alguna de ellas en particular. Los otros dos métodos serán los encargados de gestionar el movimiento de las bolas y resolver las posibles colisiones entre ellas y los bordes.

El primero de ellos es el más simple de los dos se encargará de actualizar la posición de las bolas completando los datos necesarios y llamando al método *computePhysics* de cada una de ellas.

```
private void updatePositions(float sx, float sy, long timestamp) {
    final long t = timestamp;
    if (mLastT != 0) {
        final float dT = (float) (t - mLastT) * (1.0f / 1000000000.0f);
        if (mLastDeltaT != 0) {
            final float dTC = dT / mLastDeltaT;
            final int count = mBalls.length;
            for (int i = 0; i < count; i++) {
                Particle ball = mBalls[i];
                ball.computePhysics(sx, sy, dT, dTC);
            }
            mLastDeltaT = dT;
        }
        mLastT = t;
    }
}
```

Fig. 92. Método *updatePosition* de *ParticleSystem*.

En el segundo método, *update*, será el encargado de llamar al método anterior una vez hecho esto deberá resolver las colisiones existentes entre las diferentes bolas.

```

public void update(float sx, float sy, long now) {
    updatePositions(sx, sy, now);
    final int NUM_MAX_ITERATIONS = 10;

    boolean more = true;
    final int count = mBalls.length;
    for (int k = 0; k < NUM_MAX_ITERATIONS && more; k++) {
        more = false;
        for (int i = 0; i < count; i++) {
            Particle curr = mBalls[i];
            for (int j = i + 1; j < count; j++) {
                Particle ball = mBalls[j];
                float dx = ball.mPosX - curr.mPosX;
                float dy = ball.mPosY - curr.mPosY;
                float dd = dx * dx + dy * dy;
                if (dd <= sBallDiameter2) {
                    dx += ((float) Math.random() - 0.5f) * 0.0001f;
                    dy += ((float) Math.random() - 0.5f) * 0.0001f;
                    dd = dx * dx + dy * dy;
                    final float d = (float) Math.sqrt(dd);
                    final float c = (0.5f * (sBallDiameter - d)) / d;
                    curr.mPosX -= dx * c;
                    curr.mPosY -= dy * c;
                    ball.mPosX += dx * c;
                    ball.mPosY += dy * c;
                    more = true;
                }
            }
            curr.resolveCollisionWithBounds();
        }
    }
}

```

Fig. 93 Método *update* de *ParticleSystem*.

El proceso de resolver posiciones se repetirá diez veces para asegurar que la resolución de unas produzca otras, aumentar el número de estas podría conllevar la pérdida de fluidez de la simulación. La idea es sencilla, se elige una bola y se compara su posición con el resto si con alguna de ellas sufre una colisión la forma de resolver está dando una nueva posición con un cierto toque aleatorio y actualizando estos nuevos valores en las bolas una vez que una bola comprueba con todas las demás, se comprueba la colisión con los bordes.

Por lo tanto ya conocemos los elementos clave de la simulación, ahora solo falta entender cómo se construye esta, como se obtiene los datos que la hacen posible y finalmente como estos datos pasan a dar una representación como la final.

A lo largo de los siguientes párrafos se estudiarán los siguientes elementos (Constructor y métodos) de la clase *simulationView* para poder entender cómo funciona la aplicación base.

En el constructor se le pasará el contexto donde se ejecutará el view, el cual se pasará al constructor de la súper clase, la siguiente tarea que se realiza es la obtención de un acceso al sensor del acelerómetro para poder posteriormente obtener información de este.

```

public SimulationView(Context context) {
    super(context);
    mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

    DisplayMetrics metrics = new DisplayMetrics();
    getWindowManager().getDefaultDisplay().getMetrics(metrics);
    mXDpi = metrics.xdpi;
    mYDpi = metrics.ydpi;
    mMetersToPixelsX = mXDpi / 0.0254f;
    mMetersToPixelsY = mYDpi / 0.0254f;

    Bitmap ball = BitmapFactory.decodeResource(getResources(), R.drawable.ball);
    final int dstWidth = (int) (sBallDiameter * mMetersToPixelsX + 0.5f);
    final int dstHeight = (int) (sBallDiameter * mMetersToPixelsY + 0.5f);
    mBitmap = Bitmap.createScaledBitmap(ball, dstWidth, dstHeight, true);

    Options opts = new Options();
    opts.inDither = true;
    opts.inPreferredConfig = Bitmap.Config.RGB_565;
    mWood = BitmapFactory.decodeResource(getResources(), R.drawable.wood, opts);
}

```

Fig. 94. Constructor de *SimulationView*.

Igualmente se obtiene la dimensión en centímetros de la pantalla, los cuales son pasados a pixeles, una vez realizado esto se crea la instancia de cada una de las bolas haciendo escala respecto el tamaño de la pantalla, para ello primero se definen que son un elemento *Bitmap* y se utiliza el método de *decodeResources* de *BitmapFactory* para así tener una versión *Bitmap* de un elemento externo, una vez hecho esto se escalan sus 0,5 cm respecto el tamaño de la pantalla y se almacenan este elemento en la variable *mBitmap* por otro lado se hace lo mismo con el fondo de madera de la aplicación.

Antes de realizar el método *onDraw* se realiza una tarea que será clave para el correcto funcionamiento de la aplicación que será el cálculo del centro del eje de coordenadas de la pantalla y los limites donde podrán llegar las bolas, para ello se utiliza el método *onSizeChanged* el cual se ejecutará después del constructor.

```

protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    mXOrigin = (w - mBitmap.getWidth()) * 0.5f;
    mYOrigin = (h - mBitmap.getHeight()) * 0.5f;
    mHorizontalBound = ((w / mMetersToPixelsX - sBallDiameter) * 0.5f);
    mVerticalBound = ((h / mMetersToPixelsY - sBallDiameter) * 0.5f);
}

```

Fig. 95. Método *onSizeChanged* de *SimulationView*.

Como se observa, se asegura que en el cálculo de los límites del View las bolas no salgan en ningún momento, teniendo en cuenta su diámetro en el cálculo de este último.

Junto a los elementos definidos en el constructor el método mas importante para la representación grafica e interpretación de los movimientos por la pantalla es el método *onDraw* y se ejecuta de manera cíclica detectando así los cambios que se produzca en las variables que controla una vez se asigne en el contexto de la activity.

```

protected void onDraw(Canvas canvas) {
    canvas.drawBitmap(mWood, 0, 0, null);

    final ParticleSystem particleSystem = mParticleSystem;
    final long now = mSensorTimeStamp + (System.nanoTime() - mCpuTimeStamp);
    final float sx = mSensorX;
    final float sy = mSensorY;

    particleSystem.update(sx, sy, now);

    final float xc = mXOrigin;
    final float yc = mYOrigin;
    final float xs = mMetersToPixelsX;
    final float ys = mMetersToPixelsY;
    final Bitmap bitmap = mBitmap;
    final int count = particleSystem.getParticleCount();
    for (int i = 0; i < count; i++) {
        final float x = xc + particleSystem.getPosX(i) * xs;
        final float y = yc - particleSystem.getPosY(i) * ys;
        canvas.drawBitmap(bitmap, x, y, null);
    }
    invalidate();
}

```

Fig. 96 Método *onDraw* de *SimulationView*.

La primera tarea de este cargar el fondo del lienzo, en este caso el fondo de la aplicación, lo siguiente es registrar el valor del tiempo actual desde la toma de medidas y la información obtenida de los sensores, en ese momento actualizamos los valores del *particleSystem* el cual asegurará que cada bola actualiza sus valores y resolverá las colisiones como ya se explico. Una vez realizado esto se representa la posición de cada una de las bolas siendo el centro de la pantalla el punto principal y pasando las medidas de centímetros a pixeles.

En los métodos *onResume* y *onPause* de la activity se vio como se llaman a dos métodos de *SimulationView*, *StartSimulation* y *stopSimulation*.

```

public void startSimulation() {
    mSensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_UI);
}

public void stopSimulation() {
    mSensorManager.unregisterListener(this);
}

```

Fig. 97. Métodos *startSimulation* y *stopSimulation* de *SimulationView*

El primero registra como consumidor de la información del acelerómetro al *SensorEventListener* que representa nuestra clase, para ello se llama al método *registerListener* del *sensorManager* de la activity pasando como argumentos, la referencia de la clase, el sensor del que deseamos la información y el ritmo al que se desea que esta se produzca, en este caso se utiliza la variable definida *SensorManager.SENSOR_DELAY_UI* que está indicando una frecuencia lenta de actualización, esto provocara un menor consumo de batería y será suficiente para la simulación. El segundo método simplemente desregistrará y por lo tanto parará la simulación.

2.3.3 Sensores

Por último, falta por presentar como se obtienen los datos de los sensores, como ya se comentó se debe tener un *SensorEventListener* además de haber obtenido una instancia del administrador de sensores (*SensorManager*) para así conseguir a través de él una instancia del sensor que se deseará utilizar y finalmente registrar este sensor para que el *SensorEventListener* reciba la información.

Para recibir la información se ofrecen dos clases en *SensorEventListener* la primera de ellas es *onAccuracyChanged* se encarga de detectar los cambios en el sensor, como por ejemplo si hubiera un cambio de la precisión de medida como se observó en la figura 95, el otro método a estudiar es *onSensorChanged* (*SensorEvent event*).

```
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER)
        switch (mDisplay.getRotation()) {
            case Surface.ROTATION_0:
                mSensorX = event.values[0];
                mSensorY = event.values[1];
                break;
            case Surface.ROTATION_90:
                mSensorX = -event.values[1];
                mSensorY = event.values[0];
                break;
            case Surface.ROTATION_180:
                mSensorX = -event.values[0];
                mSensorY = -event.values[1];
                break;
            case Surface.ROTATION_270:
                mSensorX = event.values[1];
                mSensorY = -event.values[0];
                break;
        }

    mSensorTimeStamp = event.timestamp;
    mCpuTimeStamp = System.nanoTime();
}
```

Fig. 98. Método *onSensorChanged* de *SimulationView*.

Lo primero que se hace es comprobar que el generador de información es el que nos interesa, hay que tener en cuenta que en caso de tener más de un sensor, toda la información de ellos pasaría por este mismo método. A la hora de obtener las mediciones hay que tener en cuenta la posición del dispositivo y para ello se utilizó el objeto que se construyó en el método *onCreate* de la actividad, el cual es capaz de detectar los cambios de orientación del dispositivo, si por ejemplo la orientación de la pantalla está definida como horizontal y se tiene encima de una superficie plana, estaríamos en *ROTATION_0* si se pusiera boca abajo el dispositivo sería *ROTATION_180* y por el contrario si se levantara perpendicular a los laterales se entraría en *ROTATION_90* y *ROTATION_270*, por lo tanto una vez obtenidos los valores necesarios para la simulación ajustaremos su signo respecto a la orientación del dispositivo. Además se obtiene el tiempo en nano segundos de cuando se obtuvo la información y de igual manera se obtiene el tiempo actual del sistema.

2.4 AcBlueServer y AcBlueClient.

Hasta ahora se ha presentado una aplicación que era capaz de tomar los datos de un sensor y representar como afectan estas a un conjunto de bolas, a partir de esta aplicación se construirán dos aplicaciones donde una de ellas facilitará la información del sensor a otra que no utilizara la información de su sensor sino la que le mandará el primero a través del Bluetooth.

Para la realización de esto se añadirá a la aplicación original, un menú de opciones emergentes, gestión del bluetooth y comunicación a través de este, una segunda activity y manejo de hilos. Inicialmente el desarrolla de estas dos nuevas aplicaciones será idéntico, solo al final se presentaran los cambios en una y otra.

2.4.1 Menú de opciones.

Muchas aplicaciones ofrecen la posibilidad de tener un menú de opciones/ajustes que se despliega al pulsar el botón de mismo nombre del dispositivo, para ello, Android ofrece una serie de métodos con los cuales realizar esta tarea de la manera más sencilla. Para poder realizar esto, primero se debe crear el archivo xml correspondiente para la representación grafica de este, por norma general todos los menús se deberían crear en la subcarpeta *menu* de la carpeta *res*, aun así al crear el primero elemento menú este nos generara su carpeta. Nuestro objetivo será obtener un menú de opciones como el siguiente.



Fig. 99. Menú de opciones.

Como se ve, se dispone de un menú con tres posibles opciones, los cuales actuaran como si fueran botones, aunque sobre su comportamiento se explicará una vez se explique cómo crear el correspondiente xml.

Lo primero será crear un nuevo elemento, en este caso un archivo Android XML File, al pulsar Next, se desplegará la siguiente ventana.

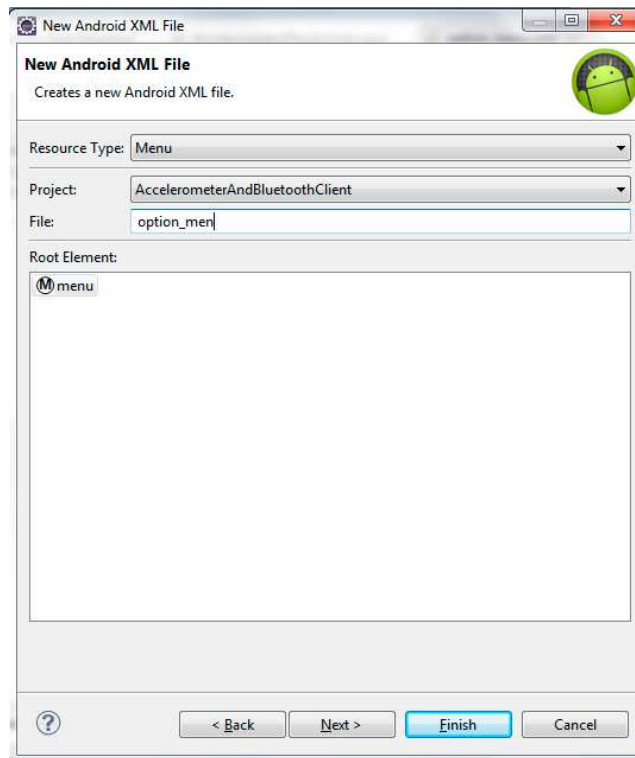


Fig. 100. Creación de un nuevo XML File.

En la primera pestaña se deberá indicar el tipo de recurso, en nuestro caso *Menu*, el proyecto al cual se desea asignar, se indica un nombre para el archivo y una vez terminado se pulsa *Finish*. En ese momento, se desplegará una ventana como la siguiente.

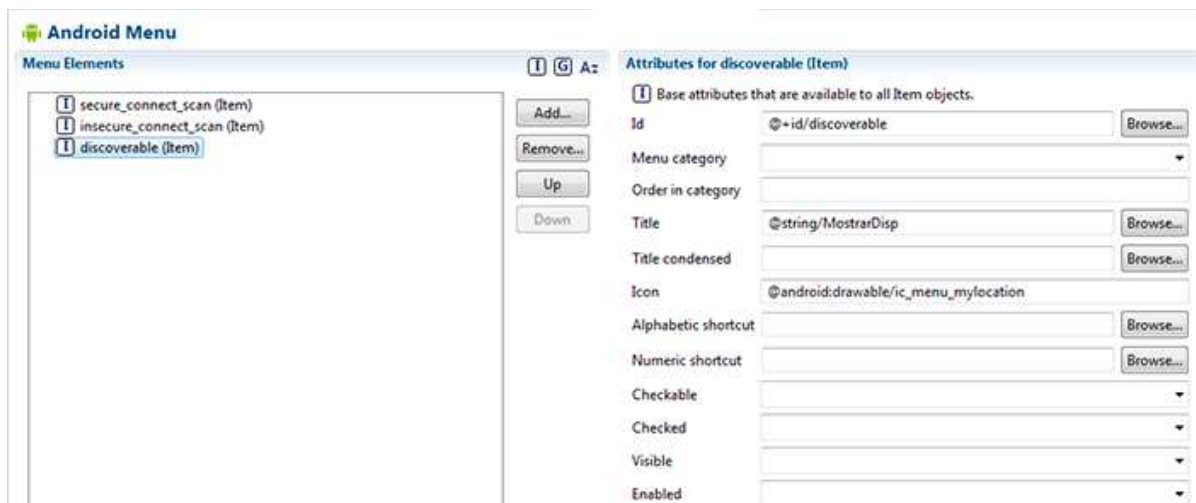


Fig. 101. Ventana de creación de un Android Menu.

Cada opción se representa como un ítem, con sus respectivas opciones, para crear uno nuevo se deberá pulsar Add y para añadirlo una vez creado Up, como se observa en la imagen con tres opciones en cada ítem será suficiente para nuestra representación, el primero su id, necesario para poder utilizarlo en la clase, el segundo la cadena de texto que se desea mostrar y por último un icono, para conseguir un icono existen cantidad definidos, para saber cuáles son podemos hacer accediendo a la pagina http://developer.android.com/guide/practices/ui_guidelines/icon_design_menu.html

2.4.2 onCreateOptionsMenu.

Volviendo a la activity principal ahora será muy sencillo añadirle este menú de opciones, para ello se deberá implementar el método *onCreateOptionsMenu* (*Menu menu*) y con la siguiente implementación para que este sea visible, para leer la información que nos daría pulsar cada uno de los botones se debe implementar el método *onOptionsItemSelected* (*MenuItem menu*), el cual se explicará después del siguiente punto.

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.option_menu, menu);
    return true;
}
```

Fig. 102. *onCreateOptionsMenu*

2.4.3 Creación de una segunda activity.

Es muy común en cualquier aplicación tener más de una activity, para ello se deberá de hacer tres tareas, la primera de ellas crear su layout, la segunda, y más lógica, es crear el .java correspondiente con su clase que herede las propiedades de la clase activity y finalmente registrar la activity en *AndroidManifest.xml*, el resultado final será el siguiente.

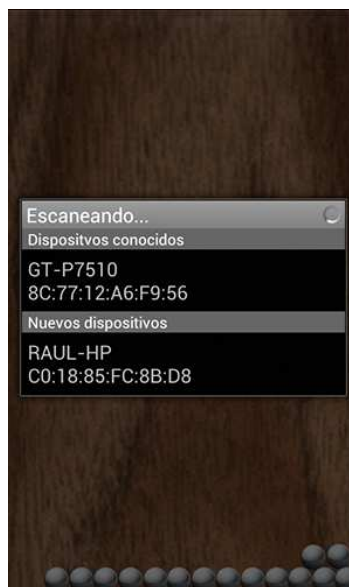


Fig. 103. Activity *VentanaDeControl*

Como se puede observar es como una ventana emergente desde la cual buscar dispositivos Bluetooth.

2.4.3.1 Creación de layout de la segunda activity.

Aunque inicialmente solo se ve un dispositivo conocido y otro nuevo, se dispone dos *ListView* y además un botón de *buscar dispositivos* que se hace invisible una vez es pulsado.

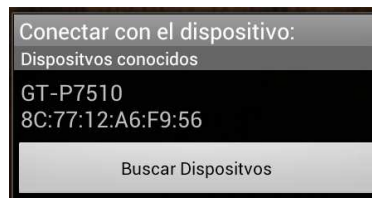


Fig. 104. Ventana de Control con Botón de Buscar Dispositivos

Además cada dispositivo está representado en un *TextView* descrito en un archivo independiente al principal para así dar a todos los elementos seleccionables la misma funcionalidad sin tener que ir de uno en uno, por lo tanto la creación del layout deberá ser similar a la siguiente.

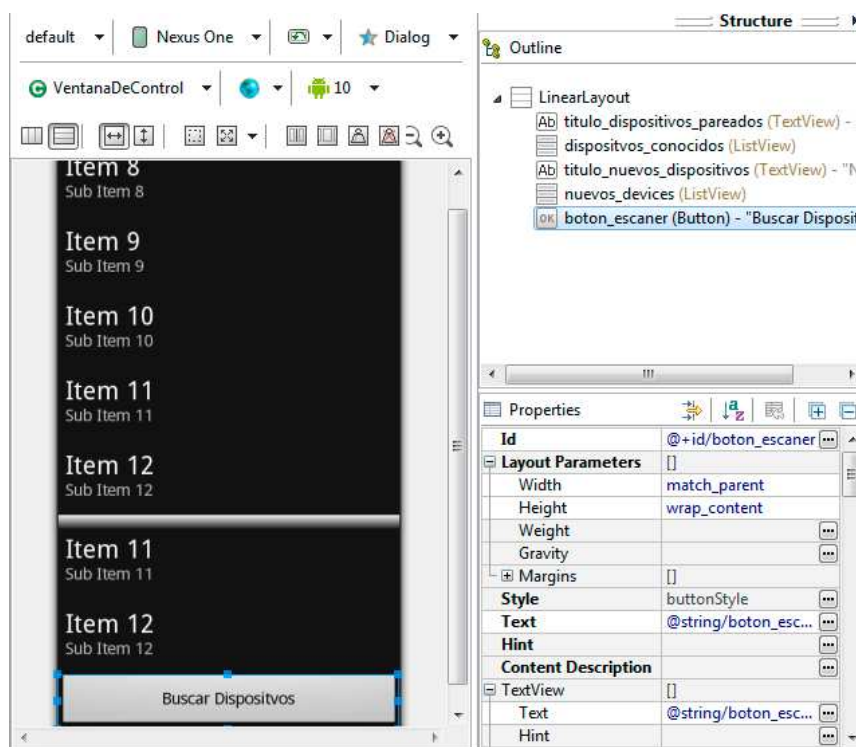


Fig. 105. Creación del layout de la activity *VentanaDeControl*

El código integro de los archivo se puede encontrar en el anexo X donde se pueden ver como se han creado cada uno de los elementos.

2.4.3.2 Clase Ventana de Control

Como se puede observar en la imagen la ventana de control solo dispone de cuatro atributos, uno un String estático que se explicara su uso posteriormente, el siguiente un *BluetoothAdapter* desde el que controlar el adaptador Bluetooth del dispositivo y finalmente dos *ArrayAdapter<string>* para enumerar los dispositivos disponibles.

```
public class VentanaDeControl extends Activity {  
  
    public static String EXTRA_DEVICE_ADDRESS = "direccion_dispositivo";  
  
    private BluetoothAdapter miBt;  
    private ArrayAdapter<String> misDispositivosConocidos;  
    private ArrayAdapter<String> NuevosDispositivos;  
}
```

Fig. 106. Inicio de la clase *VentanaDeControl*.

Además de los elementos del ciclo de vida existen otros elementos que serán claves en el desarrollo de esta actividad, que son el métodos *BuscarDispositivo*, y otros dos objetos uno un *OnItemClickListener* y otro un *BroadcastReceiver*.

El método *BuscarDispositivo* es el ejecutado al pulsar el botón, el se encargará de mostrar el aro de progreso en el titulo y cambiar su texto, y también de mostrar el titulo de nuevos dispositivos y de que nuestro adaptador Bluetooth haga el descubrimiento de otros adaptadores visibles.

```
private void BuscarDispositivos(){  
  
    setProgressBarIndeterminateVisibility(true);  
    setTitle(R.string.escaneando);  
  
    findViewById(R.id.titulo_nuevos_dispositivos).setVisibility(View.VISIBLE);  
  
    if(miBt.isDiscovering()){  
        miBt.cancelDiscovery();  
    }  
  
    miBt.startDiscovery();  
}
```

Fig. 107. Método *BuscarDispositivos*

2.4.4 Intent Filter y BroadcastReceiver.

Los Intent Filter son un subconjunto de la clase Intent, la clave de estos es que no se interactúa directamente con el llamado desde ellos sino que son una herramienta para poder filtrar los mensajes que coincidan con una determinada etiqueta, la forma más sencilla de entender su funcionamiento es ver la implementación que se hace en la actividad *VentanaDeControl*.

```
miBt = BluetoothAdapter.getDefaultAdapter();
```

Fig. 108. Acceso al adaptador local del Bluetooth

Una vez que tenemos acceso al dispositivo local del bluetooth y se ha pulsado el botón de buscar dispositivos se inicia un proceso interno el cual produce una serie de eventos y resultados que interesa tener controlados para poder realizar nuestra tarea, para ello se utiliza los Intent Filter, cada vez que el adaptador bluetooth realiza una tarea este comunica al sistema mediante un Intent lo que está realizando, nuestro objetivo será poder consumir estos Intent. En nuestra activity, interesan dos tipos de ellos, cada que se descubra un nuevo dispositivo bluetooth y también cuando se termine la búsqueda de nuevos dispositivos.

```
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
this.registerReceiver(receptor, filter);

filter = new IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
this.registerReceiver(receptor, filter);
```

Fig. 109. Uso *IntentFilter* en *onCreate* de la activity *VentanaDeControl*

Como se observa en la imagen superior, lo primero que se hace es crear un intent filter usando el constructor al que le pasamos como argumento de entrada la acción (intent) que nos interesa, lo siguiente es registrar nuestra activity usando el *BroadcastReceiver receptor* y el intent filter, en ese momento todos los intent lanzados desde el adaptador bluetooth al sistema también serán atendidos en *receptor* que deberá implementar el método *onReceive* como se ve a continuación.

```
private final BroadcastReceiver receptor = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

        if(BluetoothDevice.ACTION_FOUND.equals(action)){
            BluetoothDevice dispositivo = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            if(dispositivo.getBondState()!=BluetoothDevice.BOND_BONDED){
                NuevosDispositivos.add(dispositivo.getName()+ "\n" +dispositivo.getAddress());
            }
        }else if(BluetoothAdapter.ACTION_DISCOVERY_FINISHED.equals(action)){
            setProgressBarIndeterminateVisibility(false);
            setTitle(R.string.Selec_Dis);
            if(NuevosDispositivos.getCount() == 0){
                String sinDispositivos = getResources().getText(R.string.SinDispositivos).toString();
                NuevosDispositivos.add(sinDispositivos);
            }
        }
    }
};
```

Fig. 110. Implementación del *BroadcastReceiver* de la activity *VentanaDeControl*

El método *onReceive* se encarga de gestionar todos los intent en los que la activity se registra usando *registerReceiver* y pasando como argumento el objeto *receptor*, como se observa obtiene del intent que tipo de acción se realizo y se compara si fue un nuevo dispositivo o que la búsqueda termino, si encuentra nuevos dispositivos los añade al *ListView* correspondiente y si termino la búsqueda y además no se encontrará nuevos dispositivos se informaría de esto y antes se ocultaría el aro de progreso.

Esta es otra forma de usar los objetos intent cuando se vuelva a la actividad principal se explicará otro de los usos más común.

2.4.5 Clase `OnItemClickListener`

Es una de las formas de dotar a un elemento de la propiedad de ser pulsado y de gestionar el evento que genera, en este caso la gestión del evento será una de las causas de la finalización de la activity, junto al cierre normal de este.

```
private.OnItemClickListener DispositivosClickListener = new.OnItemClickListener() {
    public void onItemClick(AdapterView<?> av, View v, int arg2, long arg3){
        miBt.cancelDiscovery();

        String info = ((TextView) v).getText().toString();
        String address = info.substring(info.length()-17);

        Intent intent = new Intent();
        intent.putExtra(EXTRA_DEVICE_ADDRESS, address);

        setResult(Activity.RESULT_OK,intent);
        finish();
    }
};
```

Fig. 111. `OnItemClickListener` de la activity `VentanaDeControl`.

Se cancela cualquier búsqueda que se esté realizando, lo siguiente se extrae del `TextView` la dirección MAC del dispositivo bluetooth, se construye un intent usando la cadena definida al inicio de la activity y la dirección obtenida, se indica el resultado de la activity añadiendo el intent anterior y finalmente se finaliza la activity. Es muy importante incluir el resultado ya que esta activity será llamada para buscar un resultado.

2.4.6 Método `onCreate` y `onDestroy`

Una vez vistos todo los elementos de la activity, solo falta presentar qué función representan estos dos métodos de su ciclo de vida. El método `onCreate` se encargará de lo siguiente:

- Configurar el tamaño de la activity para que parezca una ventana emergente esto se realiza con la llamada `requestWindowFeature(Windows.FEATURE_INDETERMINATE_PROGRESS)`;
- Indicar que en caso de no salir como se explico anteriormente el resultado será el valor de la constante `Activity.RESULT_CANCEL`.
- Gestionar el botón de búsqueda de dispositivos.
- Obtener una instancia del adaptador bluetooth como se ve en la figura 106.
- Gestionar los `ArrayAdapter` para que los dispositivos conocidos o los que se obtengan de nuevas búsquedas se muestren por pantalla y se puedan pulsar y interactuar como ya se explico.
- Registrar nuestra activity como receptor de eventos del bluetooth.

Estas serían todas las tareas realizadas en *onCreate*, en el método *onDestroy* simplemente se desregistra el receptor de eventos del Bluetooth y se para cualquier búsqueda que hubiera en marcha.

2.4.7 Modificar el *AndroidManifest.xml*

Se tendrá que indicar que va haber una segunda activity y como tiene que ser tratada por el sistema operativo usando los intent-filter.

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".AccelerometerPlayActivity"
        android:label="@string/app_name"
        android:screenOrientation="portrait"
        android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".VentanaDeControl"
        android:label="@string/Selec_Dis"
        android:theme="@android:style/Theme.Dialog"
        android:configChanges="orientation|keyboardHidden" />
</application>
```

Fig. 112. Activities en el *AndroidManifest.xml*

Como se ve en la imagen, la primera activity es la primera que se definió, se indican cosas como la orientación, el nombre, que no aparecerá el título de la aplicación y además dispone de un intent-filter el cual indica que es hilo de entrada de la aplicación para el sistema e instalará el icono de la aplicación con esta referencia.

El siguiente elemento es la nueva activity que se configura como una ventana emergente de dialogo sin ningún tipo de intent-filter, es común que al utilizar el gestor de crear una nueva activity cree automáticamente una activity con el intent-filter visto, esto no provocará ningún error a primera vista pero si provocará que se creen dos iconos de la aplicación y que se pueda acceder a uno o otro indistintamente. Así que abra que tener ojo cuando se utilice el gestor.

Al igual que se añaden nuevas activity hay que añadir la solicitud de permisos de usuario para poder usar los elementos del dispositivo.

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.VIBRATE"></uses-permission>
<uses-permission android:name="android.permission.WAKE_LOCK"></uses-permission>
```

Fig. 113. Permisos de usuario en *AndroidManifest.xml*

Al implementar esto, provocará que cuando se instale la aplicación se informe al usuario que dicha utilizará y tendrá acceso a los elementos que se ven en la imagen, y si

el usuario sabiendo esto autoriza la instalación de la aplicación, ya que si no se tiene por ejemplo acceso al bluetooth todo lo que se ha implementado no servirá para nada y posiblemente provocará errores de ejecución.

2.4.8 onOptionsItemSelected.

Como se comento este método es el encargado de gestionar las pulsaciones en las diferentes opciones del menú, como se vio en la construcción del option-menu, solo tendrá que gestionar tres posibles eventos.

```
public boolean onOptionsItemSelected(MenuItem item){
    Intent serverIntent = null;
    switch(item.getItemId()){
        case R.id.secure_connect_scan:
            serverIntent = new Intent(this,VentanaDeControl.class);
            startActivityForResult(serverIntent,SOLICITUD_CONEXION_SEGURA);
            return true;
        case R.id.insecure_connect_scan:
            serverIntent = new Intent(this,VentanaDeControl.class);
            startActivityForResult(serverIntent, SOLOCITUD_CONEXION_INSEGURA);
            return true;
        case R.id.discoverable:
            BtVisible();
            return true;
    }
    return false;
}
```

Fig. 114 *onOptionsItemSelected* de la activity principal

Los dos primeros funcionan igual, crearan un nuevo intent para hacer una llamada al método *startActivityForResult(Intent intent, int request)* el cual a groso modo provocará que la activity haga su trabajo y cuando termine nos dé un resultado mediante un intent, en el punto siguiente se explicará cómo funciona este método, junto a otros dos más. La opción *discoverable* que llama al método *BtVisible()*.

El método *BtVisible* tiene como función hacer visible nuestro adaptador bluetooth durante un intervalo de tiempo de 300 segundos.

```
private void BtVisible() {
    if(miBtApater.getScanMode() != BluetoothAdapter.SCAN_MODE_CONNECTABLE_DISCOVERABLE){
        Intent visibleIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
        visibleIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
        startActivity(visibleIntent);
    }
}
```

Fig. 115. Método *BtVisible* de la activity principal

Como se observa primero se estudia si nuestro adaptador es visible para el resto, en el caso de no serlo se crea un intent y si indica la acción que se quiere realizar con el adaptador y se llama al método *startActivity* que será uno de los métodos que se explicará a continuación, que provocará que aparezca una ventana emergente

2.4.9 StartActivity, startActivityForResult y onActivityResult.

El primero de los tres métodos es utilizado para que una nueva activity haga su trabajo sin esperar la invocador ningún resultado las dos opciones que tenemos para implementar dicho método son:

- *startActivity(Intent intent)*, donde el intent describe que activity deberá empezar.
- *startActivity(Intent intent, Bundle bundle)*, en esta se añade un bundle para pasar información que puede ser necesaria en la nueva activity y que por ejemplo se podría extraer en el método *onCreate(Bundle bundle)* de la nueva activity.

Además existe los métodos *startActivities (Intent [] intents)* y *startActivities (Intent [] intents, Bundle options)* que funciona igual que las anteriores pero iniciando tantas activities como elementos del array de intents.

El segundo de los métodos funciona igual que el primero, pero a diferencia del anterior este esperara un resultado que deberá ser gestionado el método *onActivityResult* que se explicará después usando el ejemplo de nuestra activity principal. Al igual que el anterior existen dos opciones de implementación:

- *startActivityForResult(Intent intent, int requestCode)*, donde el intent describe que activity deberá empezar y la variable entera requestCode es un identificador que se utilizará posteriormente para identificar quien generó la respuesta.
- *startActivityForResult(Intent intent, Bundle bundle)*, al igual que el anterior añadimos un Bundle para pasar más información a la nueva activity.

Ninguna de estas llamadas es bloqueante, ni finalizara la activity principal en caso de querer realizar una tarea como esta se deberá implementar manualmente.

Como ya hemos indicado los dos métodos anteriores generan una respuesta desde la activity invocada cuando finalice que deberá ser atendida en el método *onActivityResult*, el cual se puede observar en la siguiente imagen.

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
        case SOLICITUD_CONEXION_SEGURA:
            if (resultCode == Activity.RESULT_OK) {
                conectarDispositivo(data, true);
            }
            break;
        case SOLOCITUD_CONEXION_INSEGURA:
            if (resultCode == Activity.RESULT_OK) {
                conectarDispositivo(data, false);
            }
            break;
        case SOLICITA_PERMISO_BT:
            if (resultCode != Activity.RESULT_OK) {
                Toast.makeText(this, "BT not enabled", Toast.LENGTH_SHORT).show();
                finish();
            }
    }
}
```

Fig. 116. onActivityResult de la actividad principal

El primer argumento que se recibe es el código *requestCode*, que se incluye en los otros métodos anteriores, *resultCode* es un valor de respuesta que se debe indicar en la activity invocada, como por ejemplo se hizo en la activity *VentanaDeControl* mediante el método *setResult*, finalmente un intent donde la activity invocada puede depositar datos necesarios para el invocador.

Repasando la clase *onActivityResult* y empezando de abajo arriba el primer caso supone la respuesta que hace el bluetooth cuando se quiere poner en marcha en el método *onStart* que ha sido implementado y que es una de las diferencias respecto a la activity principal en la aplicación original junto a los elementos anteriores.

```
public void onStart(){
    super.onStart();
    if(!miBtAdapter.isEnabled()){
        Intent enableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivityForResult(enableIntent,SOLICITA_PERMISO_BT);
    }else{
        if(miServidorBt == null) miServidorBt = new ServidorBluetooth(this, miHandler);
    }
}
```

Fig. 117. Método *onStart* de la actividad principal

Como se ve en el método comprueba que el bluetooth este encendido, en caso de no ser así lanza un *startActivityForResult* para realizar esta tarea en ese momento aparecerá una ventana emergente consultado al usuario, como la siguiente.

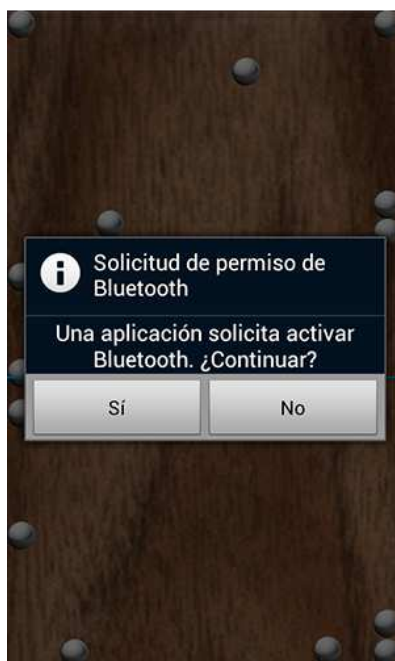


Fig. 118. Activación del bluetooth

En caso de pulsar que no provocará la finalización de la aplicación como se puede observar en la figura 114. Las otras dos opciones llaman al método *conectarDispositivo(Intent data, Boolean seguro)* el cual se puede ver a continuación.


```

private void conectarDispositivo(Intent data, boolean seguro) {
    String direccion = data.getExtras().getString(VentanaDeControl.EXTRA_DEVICE_ADDRESS);

    BluetoothDevice dispositivo = miBtAptater.getRemoteDevice(direccion);

    miServidorBt.conectar(dispositivo, seguro);
}

```

Fig. 119. Método *conectarDispositivo* de la activity principal.

Este método extrae la dirección MAC del dispositivo pasada a través del intent de *VentanaDeControl*, obtendrá un objeto de referencia del dispositivo remoto mediante un objeto *BluetoothDevice*, finalmente se llamará al método *conectar* del objeto *miServidorBt* que es de la clase implementada *ServidorBluetooth* que explicaremos seguidamente.

2.5 ServidorBluetooth

Lo primero que se estudiara será el diagrama de secuencia correspondiente al proceso de escucha, aceptación y conexión entre dos dispositivos, como se verá en ningún momento se distingue quien es cliente y quien servidor ya que tanto uno se podrá conectar a otro sea del tipo que sea, solo se distinguirán en que uno será un suministrador de información y otro un consumidor de esta una vez que estén conectados

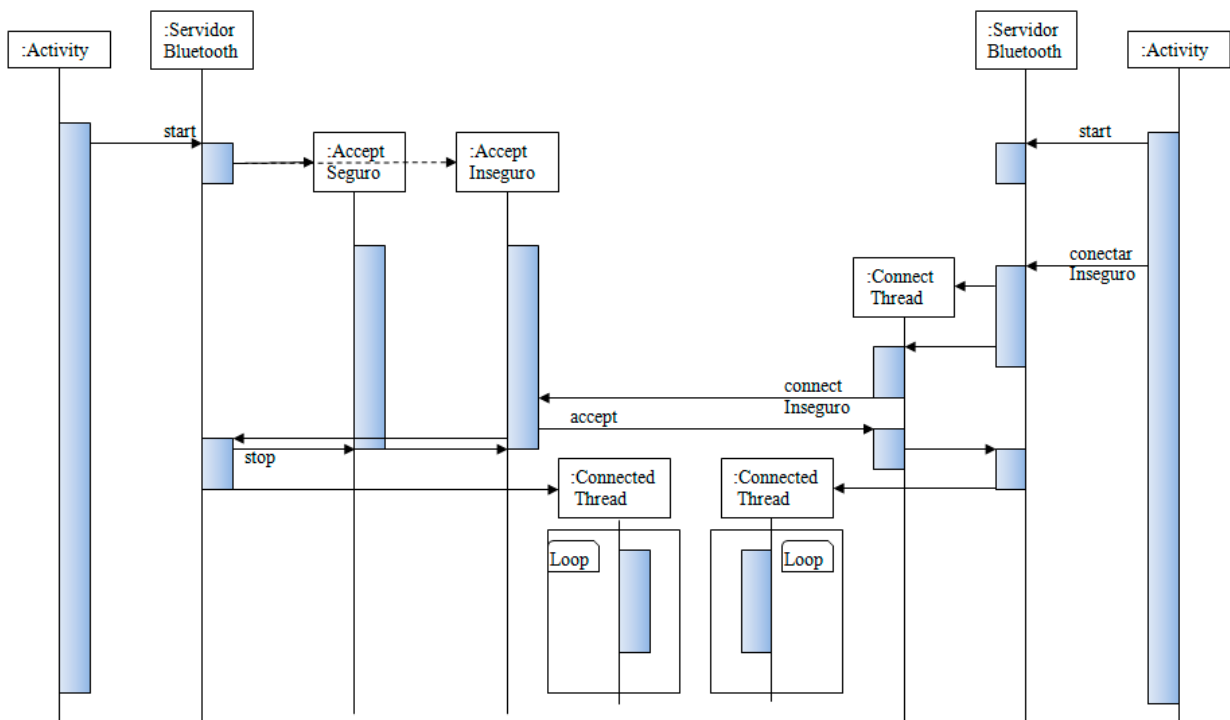


Fig. 120. Diagrama de clases

Como se observa cada activity tiene un objeto de la clase *ServidorBluetooth* el cual en un momento dado pone a trabajar, este creará dos tipos de *accept* uno de modo seguro y otro de modo inseguro, más adelante veremos las diferencias entre ambos. Por otro lado en algún momento el usuario quiere establecer conexión con otro dispositivo con un tipo de conexión, en ese momento el objeto *ServidorBluetooth* intentará establecer una conexión con el dispositivo indicado, si la conexión se logra con éxito, tanto en un lado como en el otro se finalizarían los hilos de aceptar nuevas conexiones y se iniciarían los hilos encargados del intercambio de información, los cuales estarán funcionando hasta que se pierda la conexión entre ellos, en tal caso ambos reiniciarían todo el sistema de comunicación.

2.5.1 Constructor *ServidorBluetooth*.

Antes de explicar qué función desempeña el constructor de clase lo primero es explicar qué función realizan cada una de los atributos de la clase.

```
public class ServidorBluetooth {  
  
    private static final String Nombre_Seguro = "BluetoothSeguro";  
    private static final String Nombre_Inseguro = "BluetoothInseguro";  
  
    private static final UUID Mi_UUID_SEGURO =  
        UUID.fromString("fa87c0d0-afac-11de-8a39-0800200c9a67");  
    private static final UUID Mi_UUID_INSEGURO =  
        UUID.fromString("8ce255c0-200a-11e0-ac64-0800200c9a67");  
  
    private final BluetoothAdapter miBt;  
    private final Handler miHandler;  
    private AcceptThread HiloSeguro;  
    private AcceptThread HiloInseguro;  
    private ConnectThread HiloConectar;  
    private ConnectedThread HiloConectado;  
    private int miEstado;  
  
    private boolean escuchando;  
    private boolean leer;  
  
    public static final int ESTADO_NADA = 0;  
    public static final int ESTADO_ESCUCHANDO = 1;  
    public static final int ESTADO_CONECTANDO = 2;  
    public static final int ESTADO_CONECTADO= 3;  
}
```

Fig. 121. Atributos de clase *ServidorBluetooth*

Los dos primeros String son utilizados por el *Service Discovery Protocol* del Bluetooth cuando se crea un nuevo socket, las dos variables UUID (*universally unique identifier*) que se podrían comparar con los puertos en TCP, al tener dos accept deberemos de tener uno para cada uno de ellos. Igualmente se dispone de una variable *BluetoothAdapter* para controlar el adaptador local; un manejador para poder recibir y responder mensajes desde la activity principal y todos los hilos; un hilo de cada una de las clases; una variable entera para definir el estado de la clase; un par de variables booleanas para control de los hilos y cuatro variables estáticas para definir el estado en que se encuentra el objeto *ServidorBluetooth*. Una vez conocidos los atributos de la clase, lo siguiente es conocer el constructor.

```

public ServidorBluetooth(Context context, Handler hanler){

    miBt = BluetoothAdapter.getDefaultAdapter();
    miEstado = ESTADO_NADA;
    miHandler = hanler;
}

```

Fig. 122 Constructor de *ServidorBluetooth*

Quizás una de las partes más sencillas de la clase, se obtiene acceso al adaptador local del bluetooth, se especifica que todavía no está haciendo nada mediante la variable de estado y se indica cual es el handler por el que comunicarse con el proceso de la activity.

2.5.2 Métodos de ServidorBluetooth

Se dispone de ocho métodos, nueve en el caso del servidor, encargados de gestionar el estado de la clase, los hilos y controlar los posibles sucesos, los dos primeros métodos serán utilizadas en la activity para saber el estado del objeto.

```

public synchronized int getEstado() {
    return miEstado;
}

public synchronized void setEstado(int miEstado) {
    this.miEstado = miEstado;
}

```

Fig. 123. Métodos get y set de *ServidorBluetooth*.

El siguiente método es el llamado para iniciar el trabajo de atención de conexiones mediante los hilos de accept.

```

public synchronized void start(){
    if(HiloConectar != null){
        HiloConectar.cancel();
        HiloConectar = null;
    }
    if(HiloConectado != null){
        HiloConectado.cancel();
        HiloConectado = null;
    }
    setEstado(ESTADO_ESCUCHANDO);
}

if(HiloSeguro == null){
    HiloSeguro = new AcceptThread(true);
    HiloSeguro.start();
}
if(HiloInseguro == null){
    HiloInseguro = new AcceptThread(false);
    HiloInseguro.start();
}
}

```

Fig. 124. Método start de *ServidorBluetooth*

Primero se asegura que ningún hilo este trabajando y si lo esta se para y se elimina, se apunta a null para que el recolector de basura de la maquina virtual lo haga, se indica que el estado es de escuchando nuevas peticiones y se crean los hilos encargados de aceptar nuevas peticiones y se les ponen a trabajar.

El siguiente método es el encargado de gestionar el evento de que el usuario decida conectarse a otro dispositivo.

```

public synchronized void conectar(BluetoothDevice dispositivo, boolean seguro){
    String tipo = seguro ? "segura" : "insegura";

    if(miEstado == ESTADO_CONECTANDO){
        if(HiloConectar != null){
            HiloConectar.cancel();
            HiloConectar = null;
        }
    }
    if(HiloConectado != null){
        HiloConectado.cancel();
        HiloConectado = null;
    }

    HiloConectar = new ConnectThread(dispositivo,seguro);
    HiloConectar.start();
    setEstado(ESTADO_CONECTANDO);
}

```

Fig. 125. Método conectar de *ServidorBluetooth*.

Lo primero que hacer es comprobar si se estaba llevando a cabo alguna otra conexión o si ya había alguna establecida, en ese caso cancela ambas y crea un nuevo hilo encargado de establecer la conexión, lo pone a trabajar y finalmente indica el nuevo estado.

Si el método anterior funciona correctamente y el hilo hace su trabajo lo normal es que se termine ejecutando el siguiente método.

```

public synchronized void conectado(BluetoothSocket socket,BluetoothDevice dispositivo, final String TipoDeSocket){
    if(HiloConectar != null){HiloConectar.cancel();HiloConectar = null;}

    if(HiloConectado != null){HiloConectado.cancel();HiloConectado = null;}

    if(HiloSeguro != null){HiloSeguro.cancel();HiloSeguro = null;}

    if(HiloInseguro != null){HiloInseguro.cancel();HiloInseguro = null;}

    setEstado(ESTADO_CONECTADO);
    HiloConectado = new ConnectedThread(socket, TipoDeSocket);
    HiloConectado.start();

    Message msg = miHandler.obtainMessage(AccelerometerPlayActivity.MENSAJE_DEVICE_NAME);
    Bundle bundle = new Bundle();
    bundle.putString(AccelerometerPlayActivity.NOMBRE_DISPOSITIVO, dispositivo.getName());
    msg.setData(bundle);
    miHandler.sendMessage(msg);
}

```

Fig. 126. Método conectado de *ServidorBluetooth*.

Lo primero cancelar cualquier otro hilo que estuviera trabajando, cambia el estado a *ESTADO_CONECTADO*, crea un nuevo hilo para gestionar la conexión y lo pone a trabajar, finalmente indica a la activity a través del Handler el nombre del dispositivo con el que estableció conexión.

El siguiente método deberá ser utilizado para finalizar cualquier hilo al terminar la activity ya que dejar algún proceso en marcha podría suponer una pérdida de recursos y batería del dispositivo.

```

public synchronized void stop(){
    if(HiloConectado != null){
        HiloConectado.cancel();
        HiloConectado = null;
    }
    if(HiloConectar != null){
        HiloConectar.cancel();
        HiloConectar = null;
    }
    if(HiloSeguro!= null){
        HiloSeguro.cancel();
        HiloSeguro = null;
    }
    if(HiloInseguro != null){
        HiloInseguro.cancel();
        HiloInseguro = null;
    }
    escuchando=false;
    setEstado(ESTADO_NADA);
}

```

Fig. 127. Método stop de *ServidorBluetooth*.

Simplemente cancelamos cualquier hilo que pudiera estar trabajando y se indica en el estado de la clase.

Los dos siguientes métodos son utilizados para informar cualquier posible fallo durante la realización de la conexión o durante esta.

```

private void falloConexion(){
    miHandler.obtainMessage(AccelerometerPlayActivity.MENSAJE_TOAST, -1, -1,
        "No se puede conectar con el dispositivo").sendToTarget();

    ServidorBluetooth.this.start();
}
private void conexionPerdida(){
    miHandler.obtainMessage(AccelerometerPlayActivity.MENSAJE_TOAST,
        -1, -1, "Conexión perdida").sendToTarget();

    ServidorBluetooth.this.start();
}

```

Fig. 128 Método *falloConexion* y *conexionPerdida* de *ServidorBluetooth*.

En el servidor se dispone de un método más el cual recibirá la información a enviar y la encolara para su posterior envío siempre y cuando el estado sea de conectado.

```

public void write(byte[] out){
    synchronized (this) {
        if(miEstado != ESTADO_CONECTADO) return;
        colaDeSalida.add(out);
    }
}

```

Fig. 129. Método write de *ServidorBluetooth* del servidor

Una vez controlados todos los métodos de la clase *ServidorBluetooth*, por último falta estudiar qué trabajo realizan los diferentes thread implementados.

2.5.3 AcceptThread.

Lo primero, al igual que se hizo con la clase *ServidorBluetooth*, es estudiar los atributos y el constructor de la clase.

```
public class AcceptThread extends Thread{
    private final BluetoothServerSocket miServerSocket;
    private String TipoDeSocket;

    public AcceptThread(boolean seguro){
        escuchando = true;
        BluetoothServerSocket tmp = null;
        TipoDeSocket = seguro ? "Seguro" : "Inseguro";

        try{
            if(seguro){
                tmp = miBt.listenUsingRfcommWithServiceRecord(Nombre_Seguro, Mi_UUID_SEGURO);
            }else{
                tmp = miBt.listenUsingInsecureRfcommWithServiceRecord(Nombre_Inseguro, Mi_UUID_INSEGURO);
            }
        }catch(IOException e){
            e.printStackTrace();
        }
        miServerSocket = tmp;
    }
}
```

Fig. 130. Clase *AcceptThread*.

Disponemos de dos variables una tipo servidor bluetooth será por donde se escuchará posteriormente las peticiones, además de un string para almacenar el tipo que lógicamente será *seguro* o *inseguro*.

En el constructor se comprueba el tipo de conexión que se paso como parámetro según el tipo de conexión se crean el *listener* de un tipo o de otro. La diferencia entre el modo seguro y el inseguro es que en el modo seguro se intercambian una clave de paso entre los dispositivos y que la conexión estará encriptada, y en el modo inseguro no se realiza ninguna de estas opciones. Además si el dispositivo se conecto alguna vez en modo seguro con otro, no se vuelve a pedir el intercambio de claves nuevamente.

El método run será el encargado de escuchar las peticiones y tener el accept correspondiente, como se puede ver en la siguiente imagen.

```
public void run(){
    setName("AcceptThread "+ TipoDeSocket);

    BluetoothSocket socket = null;

    while(escuchando){
        try {
            socket = miServerSocket.accept();
        } catch (IOException e) {
            e.printStackTrace();
        }
        if(socket != null){
            synchronized (ServidorBluetooth.this) {
                switch(miEstado){
                    case ESTADO_ESCUCHANDO:
                    case ESTADO_CONECTANDO:
                        conectado(socket, socket.getRemoteDevice(), TipoDeSocket);
                        break;
                    case ESTADO_NADA:
                    case ESTADO_CONECTADO:
                        try {
                            socket.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                        break;
                }
            }
        }
    }
}
```

Fig. 131. Método run de *AcceptThread*.

Se escuchan peticiones en el momento en que llega una, se comprueba si en ese momento se puede establecer o no ya que antes se ha podido establecer una conexión del otro tipo y todavía no se han cerrado los hilos de aceptar conexiones. Si el proceso fue el normal, se llama al método de *conectado* explicado anteriormente, el que como se vio cerrará cualquier hilo para aceptar conexiones.

Además se dispone del método *cancel* el cual cierra el socket y cambiar el valor de la variable boolean *escuchando* a false.

2.5.4 ConnectThread.

Es el método que gestiona la conexión cuando el usuario del dispositivo quiere conectarse a otro dispositivo, será el encargado de ejecutar el *connect* que deberá ser aceptado por un *accept* explicado en el anterior Thread. Al igual que el anterior caso primero se presentaran los atributos y el constructor de la clase.

```
public class ConnectThread extends Thread{
    private final BluetoothSocket mSocket;
    private final BluetoothDevice miDispositivo;
    private String TipoDispositivo;

    public ConnectThread(BluetoothDevice dispositivo, boolean seguro) {
        miDispositivo = dispositivo;
        BluetoothSocket tmp = null;
        TipoDispositivo = seguro ? "Seguro" : "Inseguro";

        try{
            if(seguro){
                tmp = miDispositivo.createRfcommSocketToServiceRecord(Mi_UUID_SEGURO);
            } else {
                tmp = miDispositivo.createInsecureRfcommSocketToServiceRecord(Mi_UUID_INSEGURO);
            }
        }catch(IOException e){
            e.printStackTrace();
        }
        mSocket=tmp;
    }
}
```

Fig. 132. Clase *ConnectThread*.

Se dispone de tres objetos, un *BluetoothSocket* que será utilizado para establecer la conexión y posteriormente para intercambiar la información; un *BluetoothDevice* para poder establecer el tipo de conexión para el socket y un string para indicar el tipo de dispositivo. El método Thread será muy similar al de la anterior clase. Como se puede observar en la siguiente imagen.

```

public void run(){
    setName("ConnectThread "+ TipoDispositivo);
    miBt.cancelDiscovery();
    try{
        mSocket.connect();
    }catch(IOException e){
        try{
            mSocket.close();
        }catch (IOException e2){
            e2.printStackTrace();
        }
        falloConexion();
        return;
    }
    conectado(mSocket, miDispositivo, TipoDispositivo);
}

```

Fig 133 Método run de *ConnectThread*.

Como se ve, se asigna un nombre al proceso y se cancela cualquier búsqueda que se estuviera realizando. Seguidamente se intenta realizar la llamada al método connect en caso que algo ocurriera mal, se lanzaría una excepción la cual sería atendida por el método anteriormente explicado *falloConexion()*. Si todo fue bien no se lanzará ninguna excepción y se iniciará el método *conectado*, que será el encargado de arrancar el thread que presentaremos a continuación.

2.5.5 ConnetedThread.

Esta clase esta implementada de forma diferente dependiente si la aplicación es cliente o servidor, como se puede observar.

| | |
|---|---|
| <pre> public class ConnectedThread extends Thread{ private final BluetoothSocket mSocket; private final InputStream inStream; public ConnectedThread(BluetoothSocket socket, String tipoDeSocket) { leer = true; mSocket = socket; InputStream tmpIn = null; try { tmpIn = socket.getInputStream(); } catch (IOException e) { e.printStackTrace(); } inStream = tmpIn; } } </pre> | <pre> public class ConnectedThread extends Thread{ private final BluetoothSocket mSocket; private final OutputStream outputStream; public ConnectedThread(BluetoothSocket socket, String tipoDeSocket) { escribir = true; mSocket = socket; OutputStream tmpOut = null; try { tmpOut = socket.getOutputStream(); } catch (IOException e) { e.printStackTrace(); } outputStream = tmpOut; } } </pre> |
|---|---|

Fig. 134. *ConnectThread* de cliente y servidor respectivamente.

En la imagen de la izquierda vemos la clase del cliente en la que se obtiene un flujo de entrada en el constructor a través del socket dado, en el lado servidor es igual pero en vez de un flujo de entrada será un flujo de salida. En ambos caso se ponen las variables leer y escribir a true para que se pueda realizar la tarea posteriormente.

En el lado del cliente se leerán los datos y se enviarán a la activity principal a través del handler.

```
public void run(){
    byte[] buffer = new byte[1024];

    while(leer){
        try{
            inStream.read(buffer);
            mHandler.obtainMessage(AccelerometerPlayActivity.MENSAJE_READ, -1, -1, buffer).sendToTarget();
        } catch(IOException e){
            conexionPerdida();
            break;
        }
    }
}
```

Fig. 135. Método run de ConnectedThread del cliente.

Se crea un buffer donde se almacenarán los datos leídos por el socket y una vez leído se mandan a la activity principal con la etiqueta de *MENSAJE_READ* en caso de que se cerrara la conexión en el otro extremo provocaría el fin del bucle y una llamada al método *conexionPerdida*.

En el lado del servidor el funcionamiento es similar pero sin comunicación con la activity.

```
public void run(){
    byte[] buffer;
    while(escribir){
        while(!colaDeSalida.isEmpty()){
            try {
                buffer = colaDeSalida.remove(0);
                outputStream.write(buffer);
            } catch (IOException e) {
                conexionPerdida();
                e.printStackTrace();
            }
        }
    }
}
```

Fig. 136. Método run de ConnetedThread del servidor.

Se crea un buffer de byte donde se almacenan los datos de la cola siempre y cuando esta contenga algún dato y son enviados, el resto de funcionamiento es idéntico que el del cliente.

Por último, falta estudiar las modificaciones realizadas en la activity principal, para obtener el comportamiento deseado, respecto al original.

2.5.6 Modificaciones comunes en cliente y servidor de la activity AccelerometerPlay.

Se han añadido una serie de constantes estáticas de señalización y para la gestión del bluetooth como se ven a continuación.

```
public static final int MENSAJE_CAMBIO_ESTADO = 1;
public static final int MENSAJE_READ = 2;
public static final int MENSAJE_WRITE = 3;
public static final int MENSAJE_DEVICE_NAME = 4;
public static final int MENSAJE_TOAST = 5;

private static final int SOLICITUD_CONEXION_SEGURA = 1;
private static final int SOLOCITUD_CONEXION_INSEGURA = 2;
private static final int SOLICITA_PERMISO_BT = 3;

public static final String NOMBRE_DISPOSITIVO = "nombre_dispositivo";
public static final String TOAST = "toast";
private String nombreDispositivoConectado = null;

private BluetoothAdapter miBtApatere = null;
private ServidorBluetooth miServidorBt = null;
```

Fig. 137. Modificaciones en la activity principal

El primer bloque son utilizados para el manejo de mensajes, el segundo en los intent como se explico, junto con otras herramientas para la comunicación y gestión del bluetooth. Además se ha modificado el método *onCreate* con lo siguiente.

```
miBtApatere = BluetoothAdapter.getDefaultAdapter();

if(miBtApatere == null){
    Toast.makeText(this, "No dispone de Bluetooth", Toast.LENGTH_LONG);
    finish();
    return;
}else{
    Toast.makeText(this, "Bluetooth Disponible", Toast.LENGTH_LONG).show();
}
```

Fig 138. Modificaciones en el método *onCreate*

Se comprueba de que el dispositivo dispone de adaptador bluetooth en caso de no tener la aplicación finalizaría, en caso contrario la aplicación continuará su ejecución normal, en ambos casos se comunica al usuario lo ocurrido a través de un Toast. Esta modificación esta realizada antes de asignar el view de *SimulationView* al contexto de nuestra aplicación. Además como se vio en puntos anteriores se añadido el método *onStart* del ciclo de vida donde se gestionara el encendido del adaptador. Otro método que también se añadió código es *onResume*.

```

if(miServidorBt == null) miServidorBt = new ServidorBluetooth(this, miHandler);

if(miServidorBt != null){
    if(miServidorBt.getEstado() == ServidorBluetooth.ESTADO_NADA){
        miServidorBt.start();
    }
}
}

```

Fig. 139. Modificaciones en el método *onResume*

Se comprueba que se dispone de acceso a un objeto de la clase *ServidorBluetooth* en caso de no ser así se obtiene y si está en estado de no hacer nada, se le pone a trabajar. Igualmente se ha implementado el método *onDestroy* con la única función de parar al *ServidorBluetooth* antes de finalizar la actividad principal.

Además se ha añadido un objeto Handler para gestionar los mensajes recibidos y enviados desde los diferentes *threads*.

2.5.7 Modificaciones en el servidor de la actividad *AccelerometerPlay*.

El objeto Handler mencionado anteriormente, implementa un *HandlerMessage* como se explico, los mensajes recibidos solo son informativos, cambios de estado, inicio de una nueva conexión en el *servidorBluetooth*.... El cambio clave en el servidor es, el trato con los datos obtenidos en el método *onSensorChanged*.

```

float [] infoF = new float[3];
infoF[0] = mDisplay.getRotation();
infoF[1] = event.values[0];
infoF[2] = event.values[1];

byte [] aux;
byte [] info;

aux = BytesAndFloat.toFloatToByte(infoF[0]);
info = BytesAndFloat.ConcatenarArrayByte(aux, BytesAndFloat.toFloatToByte(infoF[1]));
info = BytesAndFloat.ConcatenarArrayByte(info, BytesAndFloat.toFloatToByte(infoF[2]));

mSensorTimeStamp = event.timestamp;
mCpuTimeStamp = System.nanoTime();

info = BytesAndFloat.ConcatenarArrayByte(info, BytesAndFloat.LongToByte(mSensorTimeStamp-mCpuTimeStamp));
float relleno = 0f;
info = BytesAndFloat.ConcatenarArrayByte(info, BytesAndFloat.toFloatToByte(relleno));

if(miServidorBt != null){
    if(miServidorBt.getEstado() == ServidorBluetooth.ESTADO_CONECTADO){
        miServidorBt.write(info);
    }
}
}

```

Fig. 140. Modificaciones en *onSensorChanged* de la actividad del servidor

Además de usar los datos para la representación gráfica, se añaden los datos de las mediciones tanto de la posición de la pantalla, como los datos obtenidos del sensor y se pasan a un único array de bytes, igualmente con los valores de tiempo obtenidos, se pasa a array de bytes la resta de ellos y se concatena con el del resto para así mandar toda la información en único mensaje. Finalmente se comprueba si el *ServidorBluetooth* está en marcha y si está conectado con otro dispositivo se llama al método *write* del *ServidorBluetooth* que en cola los mensajes como se vio, ya que si no se hace así puede provocarse una sobrecarga del adaptador bluetooth con un cierre inesperado de la aplicación.

2.5.8 Modificaciones en el cliente de la activity AccelerometerPlay.

En el caso del cliente no es necesario acceder al sensor, ni a la disposición del dispositivo ya que la información para la simulación le llegara a través del bluetooth, por tanto, todos los elementos relacionados con esto han sido eliminados. Igualmente en la clase *SimulationView* ya no implementa *SensorEventListener* como se puede ver en la figura.

```
class SimulationView extends View{  
  
    private static final float sBallDiameter = 0.004f;  
    private static final float sBallDiameter2 = sBallDiameter * sBallDiameter;
```

Fig. 141. Clase *SimulationView* de la activity del cliente.

Además en el *HandleMessage* se añadido una opción más respecto del servidor en la que se leen los datos enviados desde el Thread *ConnetedThread* del *ServidorBluetooth* en ellos se descompone los elementos del mensaje original pasando estos a sus tipos de datos previos y actuando igual que el método *onSensorChanged*, como podemos ver a continuación.

```
case MENSAJE_READ:  
    byte[] readBuf = (byte[]) msg.obj;  
  
    float rotation = BytesAndFloat.byteToFloat(BytesAndFloat.desconcatenarByteArrayInf(readBuf));  
    byte[] aux = BytesAndFloat.desconcatenarByteArraySup(readBuf);  
    float [] info = new float[2];  
    info [0] = BytesAndFloat.byteToFloat(BytesAndFloat.desconcatenarByteArrayInf(aux));  
    aux = BytesAndFloat.desconcatenarByteArraySup(aux);  
    info [1] = BytesAndFloat.byteToFloat(BytesAndFloat.desconcatenarByteArrayInf(aux));  
    aux = BytesAndFloat.desconcatenarByteArraySup(aux);  
    long tiempo = BytesAndFloat.byteToLong(BytesAndFloat.desconcatenarByteArrayInf(aux));  
    mSimulationView.mTiempo = tiempo;  
    switch ((int)rotation) {  
    case Surface.ROTATION_0:  
        mSimulationView.mSensorX = info[0];  
        mSimulationView.mSensorY = info[1];  
        break;  
    case Surface.ROTATION_90:  
        mSimulationView.mSensorX = -info[1];  
        mSimulationView.mSensorY = info[0];  
        break;  
    case Surface.ROTATION_180:  
        mSimulationView.mSensorX = -info[0];  
        mSimulationView.mSensorY = -info[1];  
        break;  
    case Surface.ROTATION_270:  
        mSimulationView.mSensorX = info[1];  
        mSimulationView.mSensorY = -info[0];  
        break;  
    }  
}
```

Fig. 142. *HandleMessage* del cliente en el caso de *MENSAJE_READ*

Toda la implementación que se ha presentado, dispondrá el usuario de dos aplicaciones que pueden iniciar la comunicación de forma bidireccional y mandar datos en una única dirección, el código del proyecto original y de los nuevos cliente y servidor además de la clase para trabajar con bytes *BytesAndFloat* está disponible en el ANEXO X.

2.6 4ª Aplicación. Esqueleto de aplicación para teleoperación.

El desarrollo de la siguiente aplicación será el diseño de un esqueleto de un aplicación con la cual se podrá teleoperar dispositivos usando nuevamente la información de los sensores, igualmente se añadirán nuevos conceptos de diseño para la creación de una aplicación más interactiva con el usuario. Además dispondrá de cuatro actividades a falta de implementar la que controlaría la simulación con el sistema teleoperado, como se puede observar en la siguiente imagen.

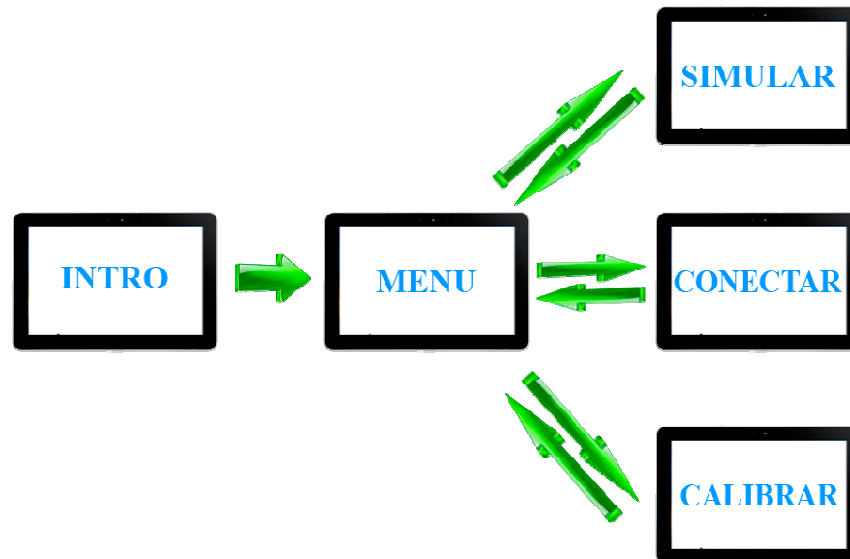


Fig. 143. Vista general de las actividades de la aplicación.

2.6.1 Creación de una activity de paso. Inicio.

Como se observa en la imagen superior, cuando el usuario ejecute la aplicación pasará por una activity que dará paso a la activity de menú, para ello se deberá crear una activity que se mostrará durante un tiempo y terminado este iniciará la siguiente activity y finalizará esta. Lo primero que se tiene que realizar es la creación del respectivo archivo xml layout.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/pantalla_inicial" >

</RelativeLayout>
```

Fig. 144. Layout de la activity Inicio.

Lo único nuevo que aporta este simple layout es el atributo *background*, en las dos primeras aplicaciones el fondo era totalmente liso, mediante este se puede añadir una imagen a la activity, para ello se deberán cargar todas las imágenes en formato *jpg* o *png* en la subcarpeta *drawable* de la carpeta *res*. Una vez visto solo falta estudiar la activity.

```

public class Inicio extends Activity {
    public static final int CALIBRAR = 1;
    public static final int CONECTAR = 2;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_inicio);
        Thread pausa = new Thread(){
            public void run(){
                try {
                    sleep(1500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally{
                    finish();
                    Intent menu = new Intent(getApplicationContext(), Menu.class);
                    startActivity(menu);
                }
            }
        };
        pausa.start();
    }
}

```

Fig. 145. Clase Inicio.

La activity inicio tiene dos objetos estáticos y finales de tipo entero que serán usados en el conjunto de la aplicación y que aprovechando la simplicidad del código de esta activity se han colocado en ella. El método *onCreate* se indica que la activity no tendrá barra de título mediante el método *requestWindowsFeature*, tras iniciar el layout se crea un hilo cuyo único trabajo será no hacer nada durante 1500 milisegundos, si esto paso con éxito se terminará la ejecución de la activity y se iniciará la siguiente activity *menu*.

2.6.2 Activity Menú.

Antes de explicar que elementos se construyen lo primero es tener una visión del resultado final que se pretende conseguir.



Fig. 146. Activity Menu en ejecución.

Como se puede observar se dispone cuatro botones en la parte central de la imagen y uno más en la parte inferior derecha, la implementación de los botones ha sido diferente a la que se ha realizado hasta ahora y que se puede ver en la siguiente imagen.

```
<ImageButton
    android:id="@+id/conectar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:contentDescription="@string/conectar"
    android:onClick="conectar"
    android:src="@drawable/conectar" />
```

Fig. 147. Creación de un ImageButton.

Como se puede ver los botones son elementos *ImagenButton*, sus atributos son comunes a los de un botón normal pero se añaden dos campos más. El *src* es donde se indica la imagen que se mostrará en el botón que al igual que la imagen fondo de la activity deberá estar ubicada en la carpeta correspondiente. Junto al *src* se deberá añadir el atributo *contentDescription* que aunque no es obligatorio pero no tenerlo provocará una advertencia en el archivo, su función es para sistemas Android que disponen de ratón es que cuando se pasa por encima indicar la función del elemento.

En la figura 142 se indico que solo se arrancarían tres activities pero en menú se dispone de dos botones más, salir y uno de información, estos provocarán la aparición de ventanas de dialogo que se explicarán en el siguiente punto.

2.6.3 Ventanas de Dialogo.

Android ofrece una herramienta muy emergente llamada *Dialog*, estos permiten mostrar una nueva ventana como se hizo en la aplicación anterior pero sin crear una segunda activity, su comportamiento estaría entre el de un Toast y un menu opciones, ya explicados, en la siguiente imagen podemos ver los dos que tendrá esta activity.



Fig. 148. Dialog en la Activity menú.

Como se observa podemos añadir botones, texto, imágenes, lista de botones... y todo esto de manera muy sencilla, como se puede ver a continuación con la implementación del *Dialog* de Información.

```

private Dialog dialogInfo(){
    AlertDialog.Builder info = new AlertDialog.Builder(this);
    info.setIcon(R.drawable.info);
    info.setTitle("Informacion:");
    info.setMessage("Autor: Raúl Velasco Gracia" +
        "\ne-mail: raul.velasco84@gmail.com" +
        "\n\nAplicación para la teleoperación de" +
        "\nun Robot Pioneer 3-AT");

    info.setPositiveButton("OK", new OnClickListener() {

        public void onClick(DialogInterface dialog, int which) {
            dialog.cancel();
        }
    });

    return info.create();
}

```

Fig. 149. Creación de *Dialog* dialogInfo.

En el objeto *Dialog* se construye un elemento *AlerDialog* el método *Builder* permite acceder al objeto desde su construcción, una vez hecho esto se añaden los atributos que se deseen mostrar, el botón se debe registrar usando el método *setPositiveButton* e implementar su método *onClick*, el cual en este caso solo cierra la ventana. Para que el dialogo sea visible se debe llamar al método *show()* como en los Toast.

2.6.4 Gestión del botón físico (o virtual) Back.

Si no se ha tratado el evento en código de la pulsación corta o larga del botón atrás este simplemente hará lo que su propio nombre indica, desde la versión 3.0 de Android se puede gestionar las dos tipos de pulsaciones que este imite con los siguientes métodos.

```

public void onBackPressed () {
    dialogoSalir().show();
}

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

```

Fig. 150. Gestión del botón back.

El primero de los métodos de la imagen gestiona la pulsación corta del botón el cual en este caso solo provocará el lanzamiento del dialogo de salir de la aplicación, al igual que el botón salir de la aplicación, el segundo detecta cualquier pulsación larga de cualquier tecla, se filtra que tecla fue pulsada como en la primera aplicación para detectar si se pulsa la que se espera si es así nuestro método iniciara el fin de la aplicación.

2.6.5 Ciclo de vida de la Activity Menú.

Una vez vistos los nuevos elementos introducidos en la activity menú, el resto han sido vistos a lo largo de esta memoria pero que pasaran a enumerar a continuación.

Se dispone de un método *onCreate* y otro *onDestroy*, la única función del primero será añadir el layout asociado y el del segundo poner fin a la ejecución. Además se dispone de cinco métodos para detectar la pulsación de todos los botones.

```
public void iniciar(View v){
    //IMPLEMENTAR PARA LA OPCIÓN SIMULAR
}

public void calibrar(View v){
    Intent calibra = new Intent(this,Calibrar.class);
    startActivityForResult(calibra, Inicio.CALIBRAR);
}

public void conectar(View v){
    Intent conectar = new Intent(this, conectar.class);
    startActivityForResult(conectar, Inicio.CONECTAR);
}

public void salir(View v){
    dialogoSalir().show();
}

public void info(View v){
    dialogoInfo().show();
}
}
```

Fig. 151. Gestión de botones de la activity menú.

Como se observa el único método que no realiza nada es el primero de ellos que se dejaría así hasta que se implementara una activity para poder teleoperar usando el resto de elementos que se verán posteriormente. Los dos siguientes son los encargados de iniciar las activities de calibrar y conectar para obtener un resultado, eso nos obligará a disponer del método *onActivityResult* como ya se explico anteriormente y que se verá a continuación, finalmente los últimos dos métodos son los encargados de mostrar los diálogos ya explicados.

La función del método *onActivityResult* es simplemente informativa de lo que paso en las otras activities como se puede ver en la siguiente imagen.

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
        case Inicio.CALIBRAR:
            if(resultCode == Activity.RESULT_OK){
                float valores [] = data.getFloatArrayExtra("valores");
                Toast.makeText(this, "valores [0] = "+ valores [0] + "\nvalores [1] = "+ valores [1], Toast.LENGTH_SHORT).show();
            }
            break;
        case Inicio.CONECTAR:
            if(resultCode == RESULT_OK){
                String address = data.getStringExtra("addr");
                Toast.makeText(this, "La dirección recibida fue "+ address, Toast.LENGTH_SHORT).show();
            }
            break;
    }
}
```

Fig. 152. *onActivityResult* de la activity *Menú*.

Simplemente se obtiene los valores que devuelven las otras activities al finalizar, el cual será mostrado mediante Toast al usuario.

2.6.6 Activity Conectar.

Es una activity muy parecida a la de la primera aplicación salvo con la diferencia que el resultado leído del teclado es de vuelta a la activity invocadora mediante el intent correspondiente al pulsar el botón volver. En la siguiente imagen se puede observar una captura de pantalla, en esta activity se deberá implementar el método de conexión del dispositivo a teleoperar.



Fig. 153. Imagen de la activity *Conectar*

Los elementos del layout han sido todos explicados, lo único que hay que tener en cuenta cuando se implemente la conexión es tener los permisos de usuarios indicados en el archivo *AndroidManifest.xml*. Y que cuando se obtuviera la conexión lo que se devolvería al invocador sería el socket de comunicación correspondiente.

2.6.7 Activity calibrar.

A diferencia de como se hizo en la anterior aplicación, aquí se dispondrá de un objeto *SensorEventListener* que será el encargado de gestionar los sensores, el propósito de esta activity es poder centrar el eje de coordenadas respecto la posición con la que el usuario dispone el terminal. Para realizar esta tarea de una manera más visual se han dispuesto de dos barras de progreso capaz de detectar el movimiento respecto los ejes como se ven en las siguientes imágenes.

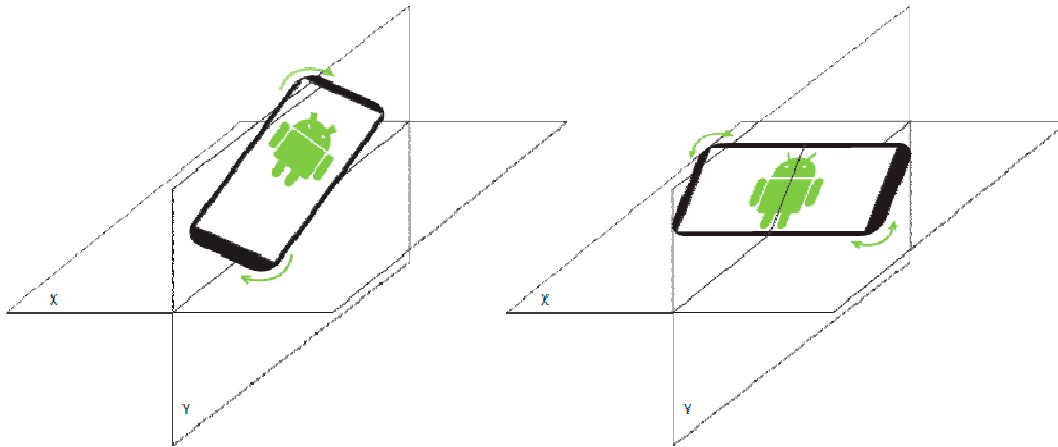


Fig. 154. Movimiento de rotación del dispositivo respecto los planos X e Y.

Teniendo en cuenta que el plano "x", es horizontal al suelo y el plano "y" perpendicular a este, las barras representaran el movimiento de estos respecto de estos planos. Cuando el móvil se mueva como se ve en la imagen de la izquierda estaremos indicando un cambio en la "dirección" y se mostrará en la barra de progreso horizontal. Cuando el movimiento es como la imagen de la derecha se indica un cambio respecto el Plano X, esta inclinación supondrá un cambio de "velocidad". Más adelante se explicara con más detalle, el resultado de la activity será el siguiente.

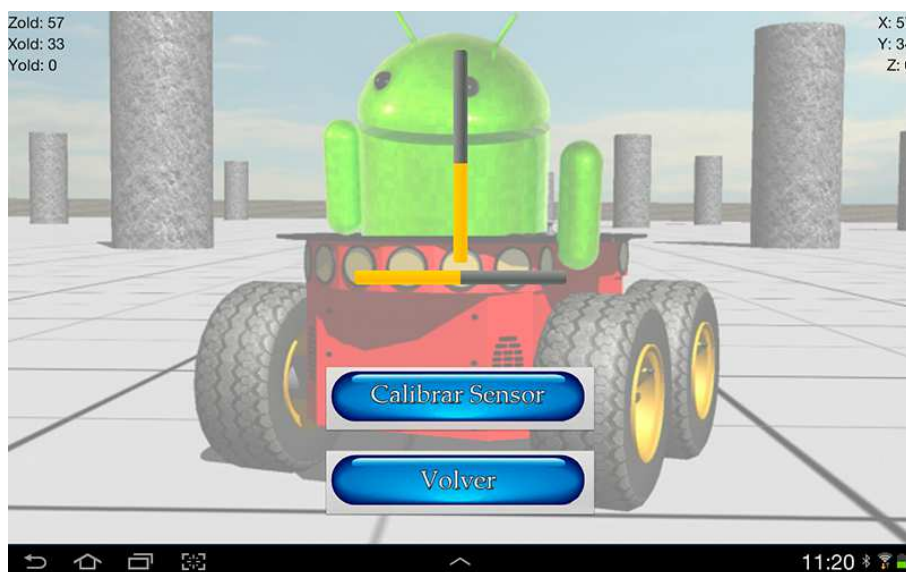


Fig. 155. Activity Calibrar.

Las barras de progreso que se han comentado y se pueden ver en la imagen anterior se agregan en el layout pero si se quieren ver de manera perpendicular como se ve, se deberá llamar al método `setRotation(float)` de un objeto `ProgressBar` debidamente asociado con el elemento del layout.

Además en las esquinas se han añadido tres `TextView` donde se mostrarán los valores de la simulación.

2.6.8 Sensor orientation Vs accelerometer y magnetic field.

El sensor `orientation` y los métodos relacionados con él fueron `deprecated` por Google, aunque la gran mayoría de desarrolladores piden que se mejore para simplificar el proceso que se tiene que llevar a cabo para obtener los mismo valores con los sensores `accelerometer` y `magnetic field`. Ya que al realizar las mediciones con los segundos se obtiene una mayor precisión y fluidez que con el primero.

Cuando se obtiene los datos del sensor `orientation` a través del método `onSensorChanged` este devuelve un array de tres elementos float, el primero de ellos indica el ángulo respecto el polo norte magnético y los otros el movimiento respecto lo planos como se ve en la figura 153, en la siguiente imagen se observa cómo se obtienen los datos de todos los sensores.

```
private SensorEventListener sensorEventListener = new SensorEventListener() {  
    public void onAccuracyChanged(Sensor sensor, int accuracy) {}  
    /* Get the Sensors */  
    public void onSensorChanged(SensorEvent event) {  
        switch (event.sensor.getType()) {  
            case Sensor.TYPE_ACCELEROMETER:  
                System.arraycopy(event.values, 0, mGravs, 0, 3);  
                break;  
            case Sensor.TYPE_MAGNETIC_FIELD:  
                System.arraycopy(event.values, 0, mGeoMags, 0, 3);  
                break;  
            case Sensor.TYPE_ORIENTATION:  
                System.arraycopy(event.values, 0, mOldOrientation, 0, 3);  
                break;  
            default:  
                return;  
        }  
    }  
}
```

Fig. 156.

Para poder obtener unos valores similares a los del sensor `orientation` pero con más precisión se deberán realizar unos cálculos matriciales y vectoriales con la información que se obtiene como se ve a continuación.

```

if (mGravs != null && mGeoMags != null) {
    boolean success = SensorManager.getRotationMatrix(mRotationM, mInclinationM, mGravs, mGeoMags);
    if (success) {
        SensorManager.getOrientation(mRotationM, mOrientation);
        X.setText("X: " + (int) (mOrientation[0]*(180/Math.PI));
        Y.setText("Y: " + (int) (-mOrientation[1]*(180/Math.PI));
        Z.setText("Z: " + (int) (mOrientation[2]*(180/Math.PI));
        Xold.setText("Xold: " + (int) mOldOrientation[0]);
        Yold.setText("Yold: " + (int) -mOldOrientation[1]);
        Zold.setText("Zold: " + (int) -mOldOrientation[2]);
    }else{
        String matrixFailed = "Rotation Matrix Failed";
        X.setText("X: " + matrixFailed);
        Y.setText("Y: " + matrixFailed);
        Z.setText("Z: " + matrixFailed);
        Xold.setText("Xold: " + (int) mOldOrientation[0]);
        Yold.setText("Yold: " + (int) mOldOrientation[1]);
        Zold.setText("Zold: " + (int) mOldOrientation[2]);
    }
}
CalibradoVelocidad();
CalibradoGiro();

```

Fig. 157. Construcción de un sensor de orientación a partir de los sensores de aceleración y medición de campos magnéticos.

Como se ve en la imagen se construye una matriz de 4x4 doubles que es almacenada en *mRotationM* esta define el entorno que en vuelve al dispositivo y que a través del método *getOrientation* construye los valores del nuevo sensor orientation, para poder comparar los nuevos valores con los antiguos se añadieron ambos para ser mostrados en los *TextView* indicados anteriormente.

Los método *calibradoVelocidad* y *calibradoGiro* son llamados al pulsar el botón calibrar con ellos se cambian la inclinación de los planos respecto los ejes originales. En la siguiente imagen se puede ver parte de uno de ellos.

```

public void CalibradoVelocidad(){
    if(!calibrado){
        Xcal = (float) (- mOrientation [1]*(180/Math.PI));
        if(Xcal<0){
            if(Xcal<=-45){
                bVertical.setProgress(90);
            }else{
                bVertical.setProgress(45-(int)Xcal);
            }
        }else{
            if(Xcal>=45){
                bVertical.setProgress(0);
            }else{
                bVertical.setProgress(45-(int)Xcal);
            }
        }
    }
}

```

Fig. 158. Método *CalibradoVelocidad* de la actividad *Calibrar*

En él se obtiene la información del sensor correspondiente, el cual se utiliza para hacer la representación a través de la barra de progreso correspondiente, cuando se pulsa el botón calibrar se memoria estas posiciones y si inicia la simulación usando este punto como referencia.

Al pulsar el botón volver lo que se manda a la actividad principal es la posición de referencia para así cuando se construya la actividad similar, el usuario pueda disponer del dispositivo como fuera calibrado en esta actividad.

CAPITULO III. ESTUDIO DEL SIMULADOR DEL ROBOT PIONEER. MOBILESIM

MobileSim es un programa empleado para la simulación de los robots de MobileRobts/ActivMedia y sus entornos, empleados para la experimentación de nuevos algoritmos. Sustituye al simulador SRIsim anteriormente distribuido con ARIA.

MobileSim se basa en el simulador Stage, con modificaciones realizadas por MobileRobts. Actualmente se puede generar entorno de simulación para los siguientes modelos de robots: p3dx, p3at, amigo-sh, amigo, powerbot, peoplebot-sh, patrolbot-sh, p2de, p2at, p3atiw-sh y seekur.

Además emplea datos de líneas de archivos de mapas de MobileRobots (.map) para simular paredes y otros obstáculos en el entorno. Para crear mapas se puede hacer uso de los programas Mapper3 o Mapper3Basic.

MobileSim ha sido probado en Windows Xp, Windows Vista, Windows 7, RedHat GNU/Linux 7.3, Debian GNU/Linux 3.1 y Ubuntu 12.04.

3.1 Instalación del simulador.

Deberá descargarlo desde la pagina web de MobileRobts o a través de la siguiente dirección <http://robots.mobilerobots.com/wiki/MobileSim> una vez que termine la descarga, ejecutamos la instalación

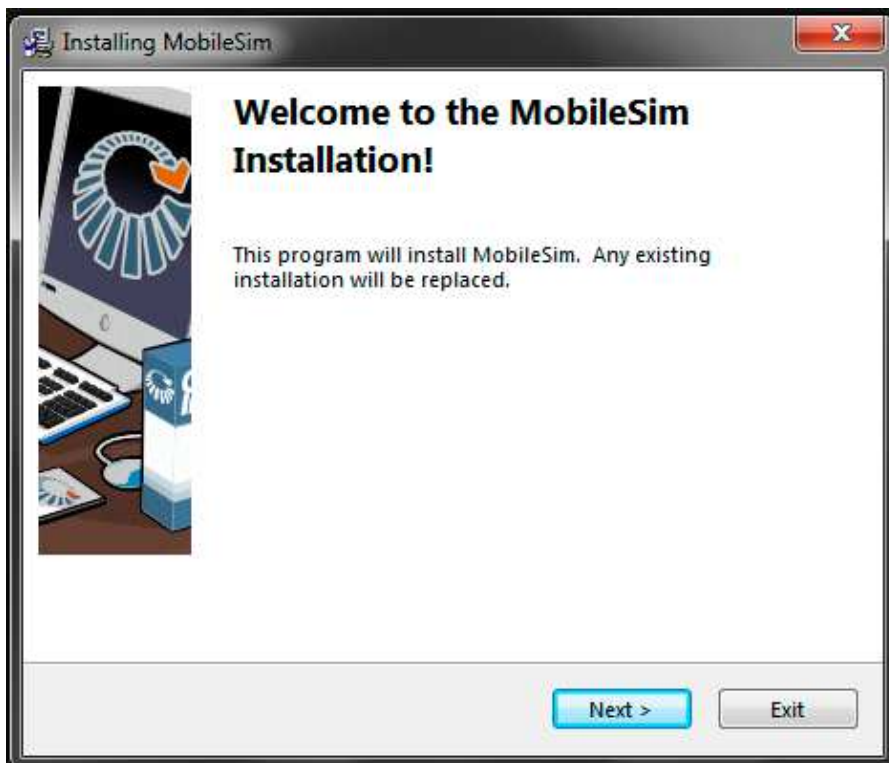


Fig. 159 Instalación del simulador MobileSim

Como se observa en la imagen se iniciará un proceso de diálogos, donde se pedirá la aceptación de la licencia de uso, todo el trabajo realizado en las anteriores versiones hasta la actual y por último indicar la ubicación en el sistema donde se quiere instalar. Tras este proceso ya tendremos instalado el simulador MobileSim.

3.2 Iniciar una simulación.

Cuando se ejecuta el simulador se ve la siguiente ventana en la cual se podrá hacer diferentes configuraciones.

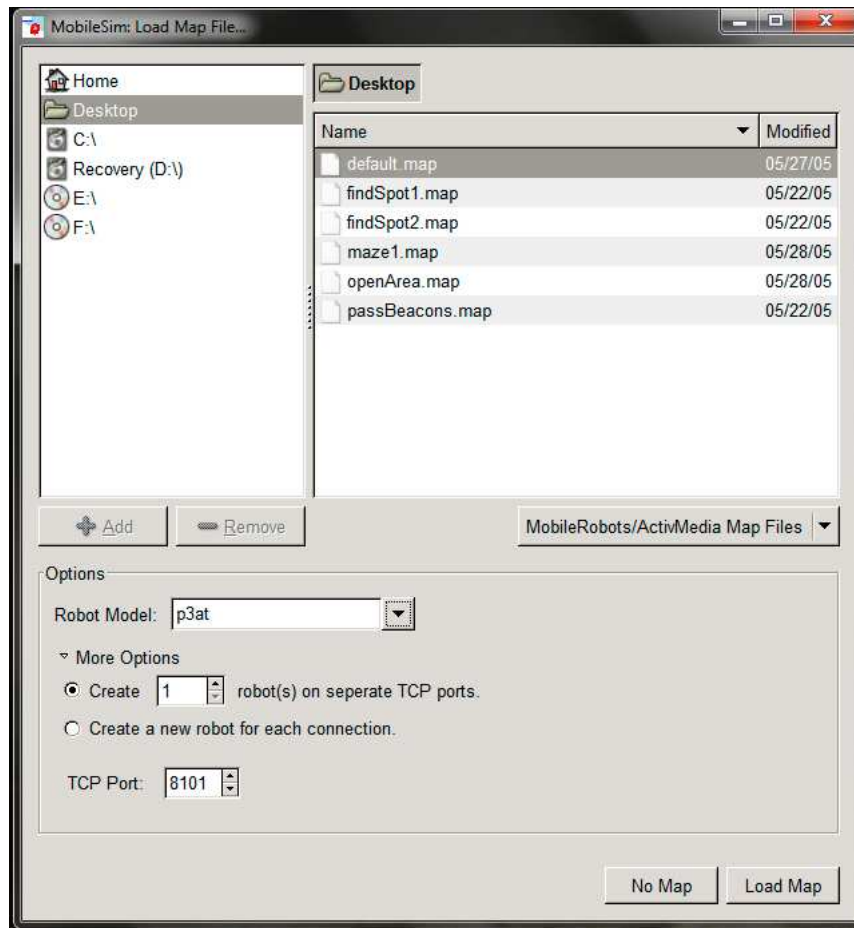


Fig. 160. Ventana de configuración de MobileSim

En esta ventana se puede indicar si se desea usar un mapa ya creado, el modelo de robot que se desea simular, la cantidad de ellos y a través de qué puerto se establece la conexión ya que el protocolo de comunicación que se usa es TCP, el simulador escucha por todas las interfaces de red las posibles solicitudes. Finalmente una vez elegida la opción deseada se indica si se desea iniciar la simulación con mapa o sin él.

La ventana de simulación es un entorno en el cual se podrá ver el comportamiento del robot a las órdenes enviadas desde un dispositivo externo al simulador, el cual mostrará las tareas que realiza de forma gráfica y aquellas que no tengan una simulación de manera textual, como se observa en la siguiente imagen.

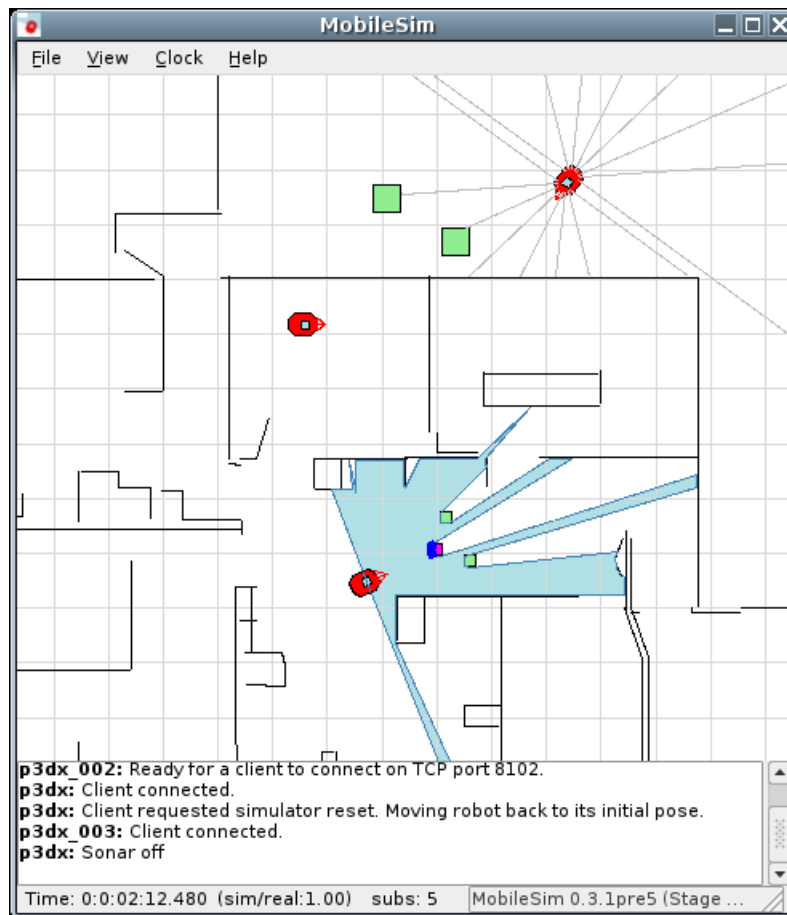


Fig. 161. Ventana de simulación de MobileSim.

En ella se puede ver a tres robots p3dx, el primero de ellos, situado en la parte superior de la imagen, siendo movido y leyendo la información del sonar, el siguiente, situado en el centro de la imagen, se encuentra desconectado como se lee en la zona de notificaciones y el último que se encuentra leyendo información del laser.

3.3 Inconvenientes del simulador.

El primer de los inconvenientes presentados es que no incluye todos los comandos de cada uno de los robots, por lo tanto algunas de las funciones de estos no podrán ser simuladas.

La otra desventaja es que aunque se simule el comportamiento del laser la información del mismo no se obtiene siguiendo el protocolo empleado por un laser real, es decir un programa que emplee el laser en el simulador necesitará usar un protocolo especial para usarlo con el robot real.

El último inconveniente aunque no relacionado directamente con el simulador, es que la gran cantidad de aplicaciones y herramientas para controlar robots están desarrolladas en lenguajes distintos de Java o Android lo cual significa que si se quieren llevar a cabo sistemas de teleoperación se deberán crear nuevas librerías para poder teleoperar con los robots desde un dispositivo móvil.

Conclusiones

Se llega al final de esta memoria y de este proyecto. Se han desarrollado tres aplicaciones dos de ellas muy y una tercera más compleja, además del esqueleto para el desarrollo de una cuarta aplicación. para un sistema operativo en pleno auge y en constante evolución. Se ha explicado desde un punto de vista actual, las técnicas y herramientas utilizadas y todo el proceso de construcción que existe al crear una nueva aplicación. Tras todo esto es el momento de sacar conclusiones de lo que se ha observado.

Facilidad de desarrollo en Android

Desarrollar aplicaciones para Android es fácil. Sin lugar a dudas, esa sería la primera conclusión que se 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 se puede 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 Apple.

La facilidad del proceso de codificación en sí mismo es una característica algo más subjetiva, puesto que depende de los conocimientos y la destreza de cada programador. Sin embargo, aquí Java parte con ventaja: no solo es uno de los lenguajes más utilizados tanto en entornos académicos como productivos, sino que de hecho, cuando apareció Android, Java llevaba ya casi 10 años siendo utilizado por los desarrolladores de aplicaciones móviles para Symbian; con lo que es muy probable que cualquier programador que se inicie en Android tenga ya experiencia previa en Java. Otros factores, como la modularidad que proporcionan las actividades o la sencillez de creación de las interfaces gráficas mediante XML, contribuyen también a esta facilidad de codificación.

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

El funcionamiento del Bluetooth.

Durante el desarrollo de este proyecto uno de los elementos que supuso un reto fueron las conexiones Bluetooth aunque existe una amplia documentación respecto a él, que no siempre funciona como se desea y que no todos los procesos de comunicación son realizados al primer intento.

Además hubo algo curioso al inicio de trabajar con el bluetooth se disponía de dos dispositivos uno con versión 3.3 y otro con una versión 4.0.3, cuando se vinculaban los dispositivos entre sí, usando como ejemplo que la versión 3.3 se vinculaba al 4.0.3, esto provocaba que la 3.3 solo pudiera ser cliente (usar connect) y si la vinculación era inversa que solo pudiera ser servidor (usar accept). Este fallo ha sido subsanado, durante el desarrollo del proyecto el dispositivo 3.3 recibió una actualización la cual soluciono este problema y permitía la conexión de una lado hacia otro indistintamente de quien se vinculara a quien inicialmente

Trabajos futuros.

Es evidente que tras el estudio de esta memoria sería ideal iniciar proyectos para la teleoperación de robots y otros dispositivos a través del esqueleto presentado.

Otra posible línea de trabajo estaría centrada en el desarrollo del meta-modelo que constituye el plugin de eclipse para el desarrollo, como por ejemplo la automatización de manera gráfica del paso de una activity a otra sin tener que entrar en los detalles del lenguaje como se permite en otros entornos de desarrollo para otras plataformas o la obligación de cuando se indica en el archivo xml que el evento de pulsar un botón será atendido por un determinado método, este se pueda autogenerar en la activity en el momento en que indica cual es su layout.

Estas son simples ideas ya que gracias a todo lo que nos ofrece Android tenemos la posibilidad de una gran abanico posibilidades para crear una gran aplicación.

Bibliografía.

- Libro: Android guía para desarrolladores de Frank Ableson, editado por Ayala.
- <http://developer.android.com/guide/components/index.html>
- <http://robots.mobilerobots.com/wiki/MobileSim>
- GSM Arena: Samsung I9100 Galaxy S II. Disponible en <http://www.gsmarena.com/samsung_i9100_galaxy_s_ii-3621.php>.
- GSM Arena: Samsung I9100 Galaxy S II. Disponible en <http://www.gsmarena.com/samsung_p7100_galaxy_tab_10_1v-3831.php>
- Libro: Java. How to Program de Deitel, Paul J. y editado por Prentice Hall.

Además de estas páginas y libros es muy sencillo encontrar ejemplos y ayuda en gran cantidad de blogs, foros, YouTube e incluso en la propia página de desarrollo de Google sobre Android.