Nuevos Paradigmas de Interacción

UNIVERSIDAD
DE GRANADA

Memoria Técnica Android

Sergio Jesús Jerez Vázquez
Carlos Lao Vizcaíno
Eduardo Muñoz del Pino
Pablo Rodríguez Fernández

Índice

Índice **2**

1. Introducción **3**

2. Diagrama de clases **3**

3. Descripción de funcionalidades. Sensores **4**
3.1 Comedor 4
3.2 Horarios 5
3.3 Mapa 5
3.4 Parking 6

4. Clases, métodos y atributos **7**
4.1 MainActivity 7
4.2 Clases asociadas al comedor 9
4.3 Clases asociadas a los horarios 11
4.3.1 ActivityHorarios 11
4.3.2 ActivityMostrarAsignatura 12
4.4 Clases asociadas al mapa 15
4.4.1 ActivitySelectRoute 15
4.4.2 ActivityMap 16
4.4.3 Podometro 19
4.5 Clases asociadas al parking 22
4.5.1 ActivityParking 22
4.5.2 ActivityTouchParking 23
4.5.3 ActivityFotoParking 24
4.5.4 ShakeDetector 25
4.6 Clases asociadas a la base de datos SQLite 26
4.6.1 IndicacionesContract 26
4.6.2 Indicación 26
4.6.3 IndicacionesDbHelper 27

5. Bibliotecas externas **29**

6. Bibliografía **29**

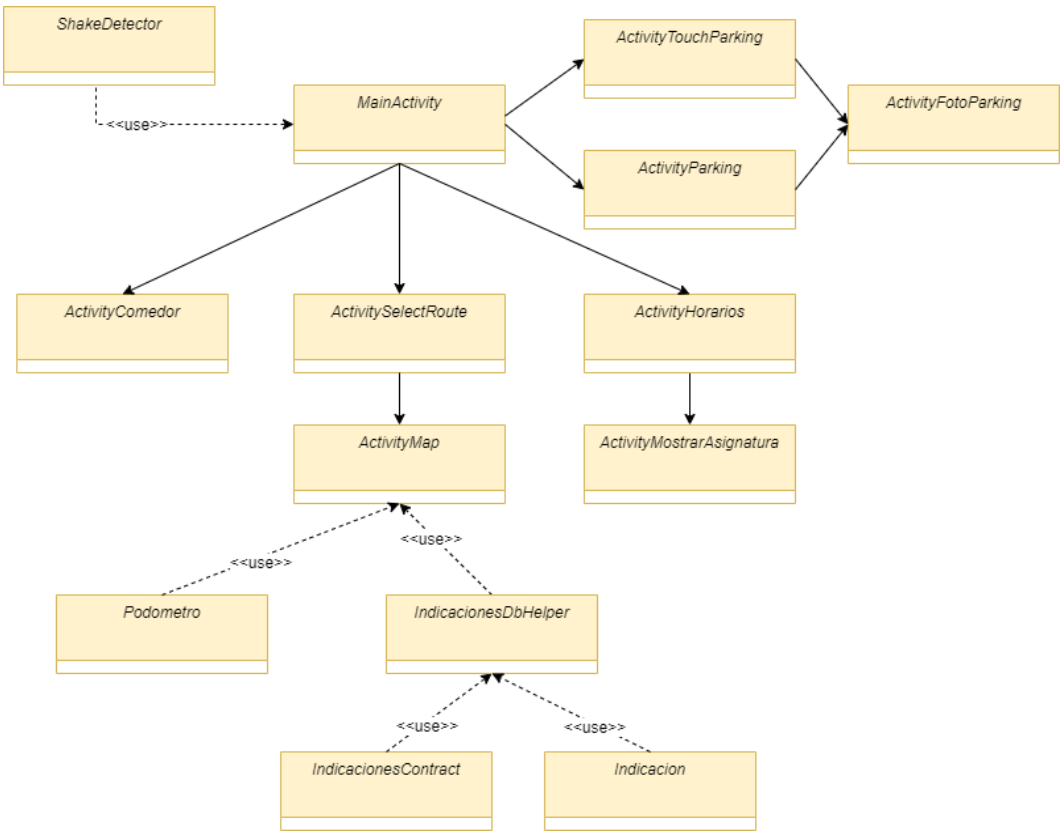
1. Introducción

En este documento se va a presentar como está hecha internamente la aplicación *ETSIIT Utilities*. Dicha aplicación está pensada para permitir, de una manera sencilla, acceder a alguna información sobre la escuela que, a día de hoy, no es posible encontrar o es de difícil acceso (a través de innumerables menús y pantallas):

- **Mapa**: guía al usuario hasta la ubicación elegida dentro de la escuela (aula, despacho,etc).
- **Horario:** Permite saber qué asignatura se está impartiendo en un aula concreta, junto con información del profesor, grupo de prácticas,etc.
- **Parking:** Permite saber, en tiempo real, las plazas de aparcamiento libres en los diferentes aparcamientos con los que cuenta la escuela (coches, motos, bicicletas y patinetes).
- **Comedor**: Permite consultar, de una manera sencilla, el menú semanal.

2. Diagrama de clases

A continuación se muestra una versión reducida del diagrama de clases de la aplicación, sin incluir atributos ni métodos, que serán explicados más adelante:



3. Descripción de funcionalidades. Sensores

3.1 Comedor

Por norma general, en cuanto al entorno gastronómico se refiere, los usuarios prefieren una buena imagen frente a la descripción de un plato de comida. Por ello, nos encontraremos con un listado gráfico del menú semanal en los comedores universitarios, con la finalidad de otorgar al usuario de una manera simple y rápida de visualizar los platos que se servirán cada uno de los días.

Sencillamente, se hace uso de la propiedad capacitiva de la pantalla de nuestro dispositivo, detectando cuántos puntos están interactuando dentro de cada uno de los días mostrados en pantalla.

De ese modo, al detectar 1 único punto de presión (1 dedo), se visualizará en pantalla el primer plato del día en el que el usuario ha pulsado. Sucesivamente, al presionar instantáneamente con 2 dedos sucederá lo mismo con el segundo plato, y con 3 dedos mostraría el postre.

A modo de introducción diremos que, a pesar de usar un único sensor en este apartado, realmente se hacen uso de 2 eventos dentro de nuestra aplicación (como veremos más adelante); y es que, por razones claramente obvias, es común trabajar únicamente con el evento que detecta una pulsación de pantalla (uso genérico de navegación), reservando el paradigma “multi-touch” en un evento de diferente índole, ya que, este último acarrea más cálculos y recursos en nuestro sistema

3.2 Horarios

Este apartado de la aplicación está pensado para que en el caso de que una persona desconozca qué asignatura se está impartiendo actualmente en un aula en particular pueda saberlo de una manera más inmersiva.

Lo primero que hay que hacer es detectar el aula de la que se está interesado conocer la información. Para ello se ha incluido un detector de textos que reconocería un texto que, hipotéticamente, debería haber colocado en cada puerta de la ETSIIT. El detector utilizado está incluido en en el **ML Kit** de Google. Además se necesitará haber dado permisos a la cámara.

Tras detectar el aula se muestra una foto en 360 grados del pasillo frente a dicho aula. En dicha foto se incluiría el aula, la asignatura, el grupo, el profesor y, opcionalmente, una foto del profesor. Para poder manejar fotos panorámicas en Android se utiliza una biblioteca externa, **PanoramaGL**, para dicho fin.

Para moverse por dicha foto se utiliza el sensor **TYPE_GAME_ROTATION_VECTOR**, el cual es similar al **TYPE_ROTATION_VECTOR**, pero sin hacer uso del sensor geomagnético.

Puesto que para este proyecto es imposible realizar fotos 360 de todas las aulas de la ETSIIT debido al tiempo evidente necesario (para obtener sólo una se han necesitado varios intentos) y, tras ello, a cada foto incluirle la información necesaria, en esta aplicación por ahora solo se detectará el aula 3.3 con la información del grupo de prácticas A1 de la asignatura Nuevos Paradigmas de Interacción.

3.3 Mapa

Una de las funcionalidades de la aplicación es el guiado a través de la facultad para llegar a una determinada ubicación (aula, despacho,etc), como realiza Google Maps en exteriores.

Ahora bien, al tratarse de un edificio, el uso del GPS es posible para realizar dicho guiado. Una alternativa es medir las distancias en pasos en lugar de en metros, sustituyendo el GPS por los sensores de detección de pasos y cuentapasos, disponibles en la mayoría de los móviles actuales.

Sin embargo, tras realizar diversas pruebas, se vió que ese enfoque no era correcto, ya que ambos sensores tienen una latencia demasiado alta como para poder guiar de una manera realista al usuario (STEP_DETECTOR tiene unos 2 segundos de latencia y STEP_COUNTER roza los 10).

Finalmente, tras explorar diversas alternativas, se encontró una manera de contar los pasos empleando el **Acelerómetro**, lo cual se detallará más adelante.

3.4 Parking

La aplicación da la opción de consultar las plazas de parking de los diferentes tipos de vehículos de los usuarios de la escuela. Por ello se da la opción de consultar por separado las plazas disponibles de **bicicletas, patinetes, coches y motos.**

Hay un menú para consultar con botones, las plazas de cada parking, y ver por separado una cámara o una foto actualizada de la situación actual de cada parking.

Como esta funcionalidad se supone que se va a usar desde un vehículo, se ha dado la opción de consultar esta información usando gestos. Para ello, desde el menú principal, si se agita el móvil entramos a una pantalla de dibujado. En dicha pantalla si dibujamos la inicial de cada parking en mayúscula (B, P, C o M), se entraría a la pantalla indicada anteriormente, en la cual se vería la cámara o foto actualizada de cada parking.

Para la funcionalidad de agitar se ha usado una clase externa creada por Bob Lee y Eric Burke llamada **ShakeDetector**, la cual hace uso del **acelerómetro**. Para la funcionalidad de dibujar las letras, se ha usado la librería **gesture**, la cual comprueba la semejanza del gesto dibujado con varios gestos previamente guardados.

4. Clases, métodos y atributos

4.1 MainActivity

Descripción

En esta clase está asociada a la actividad principal. En ella lanzan las demás actividades encargadas de dar funcionalidad a la app. También se puede lanzar la actividad Touch Parking agitando el teléfono.

Agitar para ver plazas de parking

Atributos

```
protected static IndicacionesDbHelper database;
```

Base de datos usada en la app. Se declara aquí para que sea accesible desde todas las actividades (si fuese necesario).

```
private static final int CAMERA_REQUEST = 1888;
```

Usado para obtener los permisos de la cámara.

```
private SensorManager sm;
```

Usado para almacenar el gestor de sensores para detectar la agitación.

```
private ShakeDetector sd;
```

Almacena una instancia de la clase ShakeDetector, la cual detecta, usando el *sm* si se agita el teléfono.

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Se lanza al crear la actividad. Además de asociar el layout a la actividad, inicializa la base de datos de SQLite, solicita los permisos de cámara e inicializa la clase encargada de detectar si se agita el teléfono.

```
protected void onResume()
```

Invocado cuando se resume la actividad. Lo sobreescribimos para volver a lanzar el sensor de agitación.

```
protected void onPause()
```

Invocado al salir de la actividad. En él paramos el sensor de agitación.

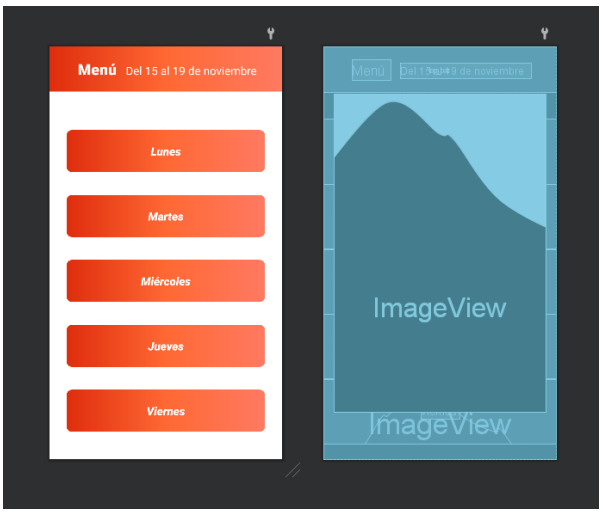
```
public void goComedor(View view)
public void goHorarios(View view)
public void goMap(View view)
public void goParking(View view)
```

Métodos usados por los botones para lanzar las actividades correspondientes.

4.2 Clases asociadas al comedor

Descripción

Para la funcionalidad del apartado de comedores, ha sido utilizado un único Activity en nuestra aplicación, ya que de otra manera, el usuario podría experimentar cierta lentitud en la navegación. Esta "ligereza" nos otorga una sensación fiel de la característica multi-touch, pues de lo contrario (como por ejemplo sucede en la navegación por un mapa… el zoom de una fotografía), estropearía la navegabilidad y daría lugar a confusiones.



Como se puede apreciar en la ilustración anterior, en el árbol de componentes de la aplicación podemos apreciar el esqueleto sobre el que ha sido desarrollado esta pantalla. De carácter relevante, es el uso adecuado de la visibilidad de los componentes, ya que pese a permanecer ocultas las capas superiores de la pantalla podemos inducir errores si no configuramos correctamente los eventos de los mismos (aunque una imagen esté visualmente oculta podría interferir con los demás elementos de la pantalla, obteniendo resultados inesperados y errores de diversos tipos)

Atributos

Las variables a usar en esta sección de la aplicación se limitan prácticamente a la interacción de múltiples contenedores de ímagenes (los *ImageView*). Concretamente, los podríamos clasificar por 2 finalidades:

- Muestra del cajetín/botón del día:

```
private ImageView box_lunes, box_martes ...;
```

Este contenedor en lugar de una imagen, adopta los estilos de diseño marcados por la línea de colores creada en nuestro tema, y sobre él colocaremos un simple componente de texto sin ningún propósito específico salvo el de indicar al

usuario el día de la semana que “contiene” en su interior. Es el encargado de alertar al sistema si detecta una pulsación dentro de sus coordenadas

- Visualización de un plato de comida:

```
private ImageView plato_1_1, plato_1_2...;
```

Del mismo modo, gracias al contenedor de imágenes, alojaremos en pantalla cada uno de los platos disponibles durante esta semana. Por defecto, a través del valor alpha genérico para todos los elementos visuales, nuestras imágenes del menú permaneceran “ocultas”. La clave aquí consistirá en activar el plato cuando se haya activado su coordenada y configuración

```
private View telon_negro;
```

Finalmente, por aumentar la claridad y mejorar el diseño de la aplicación, haremos uso de una vista de pantalla, que se interpondrá cada vez que el usuario quiera mostrar una imagen de un plato. Así, daremos la sensación de “pop-up”, realzando la imagen y dejando el fondo de la aplicación en segundo plano.

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Justo en el momento de la creación de nuestra pantalla comenzaremos a capturar cada uno de los componentes del layout que hemos creado (cajetines de los días de la semana… las imágenes ocultas… y nuestro telón negro. El uso del método “findViewById()” propio de android resulta imprescindible para esta finalidad.

Tal y como indicamos en el prólogo de este apartado, únicamente se hace uso de un listener:

```
.setOnTouchListener(new View.OnTouchListener()
```

Con este método, dotaremos a las variables pertinentes de la capacidad para detectar sobre ellas mismas la interacción en pantalla del usuario.

Para ello, tendremos que sobrecargar el método `onTouch(View, MotionEvent)` ya que el evento pasado por parámetro será del que obtendremos la información relevante sobre la pantalla.

- El parámetro `getPointerCount()` nos indicará el número de puntos presionados.
- El parámetro `getAction()`, a través de su estado (ACTION_UP) o (ACTION_DOWN) nos indicará si se hay presión o ésta es inexistente.

Con par de bloques condicionales, y un orden coherente (ya que con múltiples dedos el parámetro de ACTION_DOWN está igualmente activo). Resulta sencillo

detectar de qué manera el usuario está haciendo uso de su pantalla, y gracias a la activación del Listener en cada uno de nuestros cajetines podremos activar o desactivar la imagen del plato correspondiente.

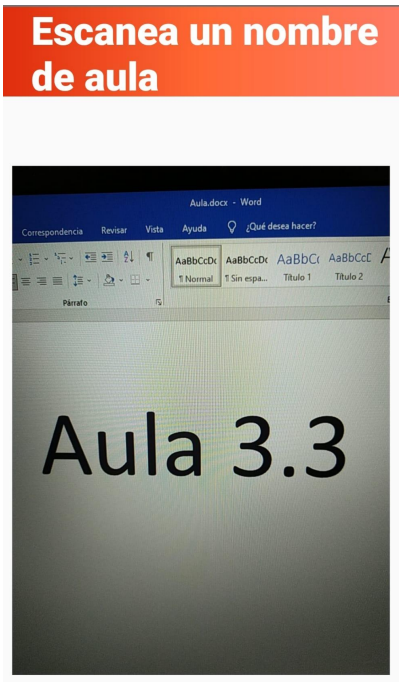
Esto último, lo hacemos a través de la función del sistema `setAlpha()` que nos permitirá cambiar el valor alpha del componente. Un valor de 0, producirá un resultado “transparente” u oculto, mientras que un valor de 1 será equivalente al 100% de opacidad.

4.3 Clases asociadas a los horarios

4.3.1 ActivityHorarios

Descripción

Dentro de esta clase se maneja la cámara, se detecta el texto y éste se procesa.



Aula 3.3

Atributos

```
private CameraSource mCameraSource;
```

Atributo necesario para manejar la cámara y el detector subyacente.

```
private SurfaceView mCameraView;
```

Necesario para mostrar lo que ve la cámara en pantalla.

```
private TextView mTextoAulaDetectada;
```

Relacionado con la interfaz gráfica, en específico un texto del aula detectada.

```
private boolean aulaEncontrada = false;
```

Booleano necesario para controlar cuando se ha detectado un aula para poder cambiar a “ActivityMostrarAsignatura”.

```
private static final int CAMERA_REQUEST = 1888;
```

Identificador necesario que hay que pasar como parámetro al pedir los permisos de la cámara.

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Invoca al constructor de la clase padre (*AppCompactActivity)* e inicializa el layout de ActivityHorarios. Finalmente llama a la función detectarTexto().

```
public void onResume()
```

Al igual que con la función anterior se llama al constructor de la clase padre, se inicializa de nuevo el booleano aulaEncontrada como “false” y se llama a detectarTexto(). Es necesaria para el caso en el que se quiera detectar otra vez un aula tras haber mostrado la información de otra aula anteriormente.

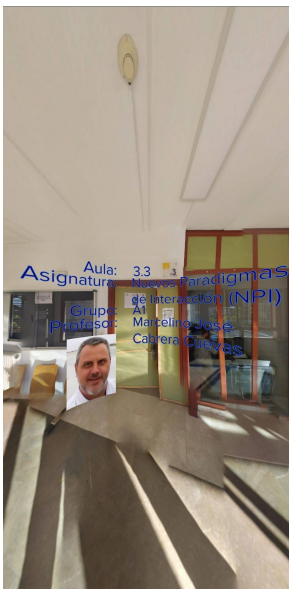
```
private void detectarTexto()
```

Dentro de este método se encuentra el código encargado de crear el detector de texto y de crear la cámara y su respectivo listener. La tarea principal, además de lo anterior, es que en el caso que detecte el texto correspondiente a un aula (“Aula 3.3”), muestra dicho texto, vuelve verdadero aulaEncontrada y cambia a “ActivityMostrarAsignatura” con un intent.

4.3.2 ActivityMostrarAsignatura

Descripción

En esta clase introducimos la foto en 360 grados a través de PanoramaGL y manejamos los datos del GAME_ROTATION_VECTOR para modificar la cámara de PanoramaGL. Esta clase además debe implementar la clase SensorEventListener para manejar sensores.



Atributos

```
private SensorManager mSensorManager;
```

Permite obtener información general de los sensores, incluyendo instancias de sensores concretos.

```
private Sensor mSensorRotationVector;
```

Instancia del sensor GAME_ROTATION_VECTOR que devuelve el SensorManager.

```
private Display mDisplay;
```

Necesario para conocer la orientación del dispositivo y para manejar ventanas personalizadas.

```
private PLManager plManager;
private ActivityMostrarAsignaturaBinding binding;
PLSphericalPanorama panorama = new PLSphericalPanorama();
```

Atributos necesarios para utilizar PanoramaGL y para modificar las vistas de la interfaz gráfica.

```
float azimuthDegrees = 0.0f;
float pitchDegrees = 0.0f;
float rollDegrees = 0.0f;
```

Componentes de la orientación obtenidos del sensor GAME_ROTATION_VECTOR en grados.

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Al igual que las otras funciones onCreate(Bundle savedInstanceState) invoca al constructor de la clase padre (*AppCompactActivity)* e inicializa el layout. Sin embargo, en vez de hacer uso de la función findViewById() para asignar los elementos del layout a nuestros atributos se hace uso de las clases Display, ActivityMostrarAsignaturaBinding, ViewGroup y WindowManager para poder crear un layout propio a través de PanoramaGL. Además se obtiene la instancia del sensor GAME_ROTATION_VECTOR del SensorManager. Además creamos e inicializamos el PLManager con todas sus funcionalidades desactivadas menos el zoom (vamos a manejar nosotros un sensor y no vamos a usar la forma predeterminada de movimientos con sensores que incluye PanoramaGL). Además cargamos la foto 360 inicializando donde mira la cámara.

```
protected void onStart()
```

Se invoca el constructor del padre y en caso de que exista el sensor se registra su “listener”.

```
protected void onStop()
```

Se invoca el constructor del padre y se deja de registrar el “listener” del sensor.

```
protected void onResume()
protected void onPause()
protected void onDestroy()
```

Se invocan los constructores del padre en cada caso. Otras funciones del ciclo de vida de Android.

```
public boolean onTouchEvent(MotionEvent event)
```

Función necesaria en caso de permitir movimiento por la imagen con las pulsaciones de los dedos. No es nuestro caso, ya que está desactivado en el PLManager.

```
public void onSensorChanged(SensorEvent sensorEvent)
```

Método necesario para la clase SensorEventListener. Aquí obtenemos la matriz de rotación de los datos que devuelve el sensor. Según la orientación del dispositivo ajustamos la matriz. De esa matriz obtenemos la dirección en un vector de tres componentes con la función getOrientation() de SensorManager. Los tres valores de orientación son el azimuth, pitch y roll (en ese orden). Los convertimos en grados y los almacenamos en los correspondientes atributos definidos en la clase. Creamos un objeto PLRotation usando como parámetros pitchDegrees, azimuthDegrees y rollDegrees (en dicho orden puesto que PanoramaGL representa la orientación de esta manera). Finalmente, se mueve la cámara con la rotación anterior.

```
public void onAccuracyChanged(Sensor sensor, int accuracy)
```

Función necesaria para poder implementar la clase SensorEventListener. Sin embargo solo está definida y no contiene nada de código.

4.4 Clases asociadas al mapa

4.4.1 ActivitySelectRoute

Descripción

Esta clase está asociada a la actividad de elegir la ruta que quiere seguir el usuario.

Atributos

```
public static final String RUTA_SELECCIONADA
```

Cadena utilizada por un Intent para pasar la información al mapa sobre ruta seleccionó el usuario.

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Invoca al constructor de la clase padre de la clase (*AppCompactActivity)* y carga el layout asociado a la clase.

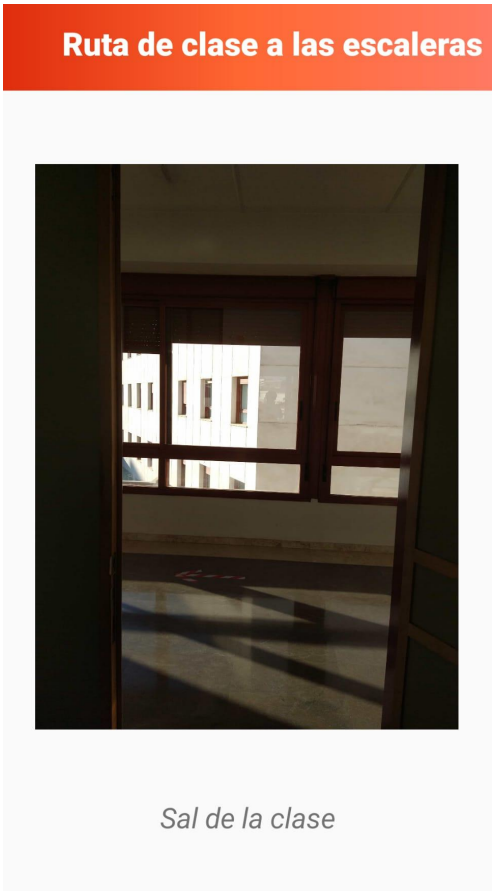
```
public void elegirRuta(View view)
```

Da paso a la actividad asociada al Mapa, envíandole a través del atributo RUTA_SELECCIONADA la clave asociada a la ruta (para saber qué información se debe cargar de la base de datos)

4.4.2 ActivityMap

Descripción

Esta clase está asociada a la actividad de guiar al usuario a través de una ruta seleccionada en *ActivitySelectRoute*. Se encarga tanto de la parte de visualización como de la obtención de los datos de la base de datos.



Sal de la clase

Atributos

```
private ArrayList<Indicacion> indicaciones
```

Almacena las indicaciones asociadas a un ruta concreta, obtenidas de la base de datos.

```
private TextView titleIndicacion
private ImageView imgMapa
private TextView textoIndicacion
```

Atributos asociados a elementos del Layout.

```
private SensorManager mSensorManager
```

Permite obtener información general de los sensores, incluyendo instancias de sensores concretos

```
private Sensor mSensorAcc
```

Instancia del sensor de aceleración, necesario para el funcionamiento del podómetro

```
private Podometro podometro
```

Instancia de la clase que realiza el conteo del número de pasos (explicada más adelante)

```
public static float mStepCounter = 0
```

Números de pasos en el momento actual. En cada cambio de indicación se reestablece a 0 (cuenta el número de pasos dentro de la indicación actual, no el número de pasos totales).

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Invoca al constructor de la clase padre de la clase (*AppCompactActivity),* carga el layout asociado e inicializa los atributos de la clase. En el caso de *Indicacion*, llama al método *inicializarIndicaciones( )* para obtenerlas. Además, registra un listener sobre el sensor de aceleración, para que se notifique cuando realice una nueva medición. Finalmente, invoca al método *inicializarIndicaciones(*ruta*),* siendo "ruta" el nombre de la ruta que se debe mostrar, obtenido del Intent enviado desde *ActivitySelectRoute*.

```
protected void onStop()
```

Este método se ejecuta cuando se cambia de actividad. Permite que se deje de escuchar el sensor de aceleración por parte de la aplicación de forma segura.

```
public void onSensorChanged(SensorEvent event)
```

Este método se invoca cada vez que el sensor de aceleración realiza una nueva medición. Invoca al método *calculateSteps ()* de la clase *Podometro* para actualizar el número de pasos del usuario.

```
private void inicializarIndicaciones(String ruta)
```

Este método obtiene las indicaciones asociadas a la ruta elegida, cargándolas desde la base de datos. Además, toma los datos de la primera indicación y muestra su información en la pantalla (como inicio de ruta).

```
private void updateIndicacion()
```

Este método comprueba si el usuario superó el número de pasos asociados a la indicación actual. De ser así, elimina dicha indicación, carga la información de la siguiente (texto e imagen) y reestablece el número de pasos de la indicación.

4.4.3 Podometro

Descripción

Esta clase es la encargada de contar los pasos que da el usuario. Para ello, toma los datos facilitados por el acelerómetro y, tras filtrarlos para contrarrestar el efecto de la gravedad, crea una serie de tiempo (juntando varias medidas consecutivas) sobre la que aplica un algoritmo de detección de picos (*peak detection*). Cada uno de estos “picos" se corresponde con un paso del usuario.

Las funcionalides de esta clase son una adaptación del repositorio de Github *isibord/StepTrackerAndroid* [1], el cual contiene el desarrollo de una aplicación para contar los pasos del usuario y mostrar diversas métricas sobre las señales procesadas.

Atributos

```
private static int SMOOTHING_WINDOW_SIZE = 20
```

Número de mediciones que se van a utilizar para alisar la señal (con el fin de eliminar falsos picos/pasos).

```
private float mRawAccelValues[] = new float[3]
```

Esta variable se utiliza para almacenar las tres componentes de la medición que proporciona el sensor de aceleración.

```
private float mAccelValueHistory[][] = new float[3][SMOOTHING_WINDOW_SIZE]
```

Este atributo almacena las últimas 20 mediciones (recordar que SMOOTHING_WINDOW_SIZE = 20) del sensor de aceleración.

```
private float mRunningAccelTotal[] = new float[3]
```

Suma de la aceleración de las últimas 20 mediciones.

[1] https://github.com/isibord/StepTrackerAndroid

```
private float mCurAccelAvg[] = new float[3]
```

Media de las últimas 20 mediciones de aceleración en cada dirección.

```
private int mCurReadIndex = 0
```

Índice de la medición de aceleración actual (dentro del array de mediciones).

```
private double mGraph1LastXValue = 0d
private double mGraph2LastXValue = 0d
private double lastXPoint = 1d
private LineGraphSeries<DataPoint> mSeries1
private LineGraphSeries<DataPoint> mSeries2
```

Estos atributos son utilizados por el algoritmo de detección de picos para almacenar las mediciones de las aceleraciones. Las clases *LineGraphSeries* y *DataPoint* pertenecen a la biblioteca *GraphView*[2], que permiten facilitar los cálculos.

```
private double lastMag = 0d
private double avgMag = 0d
private double netMag = 0d
```

Estos atributos son utilizados para poder reducir la influencia de la gravedad en las mediciones del acelerómetro.

```
double stepThreshold = 0.8d
double noiseThreshold = 1.8d
```

Estos atributos fijan los umbrales inferior y superior, respectivamente, para que un pico en la señal se considere un paso. Es decir, un valor se considerará que representa un paso solo si

stepThreshold < valor < noiseThreshold

Estos atributos fueron ajustados mediante pruebas para encontrar los que mejor se adaptan a la aplicación.

```
public static float mStepCounter = 0
```

Número de pasos detectados por el podómetro.

Métodos

[2] https://github.com/jjoe64/GraphView

```
public Podometro(Sensor SensorAcc)
```

Constructor de la clase. Asigna el sensor de aceleración (el cual se le pasa desde la clase *ActivityMap*) y crea instancias de *LineGraphSeries.*

```
public void calculateSteps(SensorEvent event)
```

Función que realiza el filtrado de la serie de tiempo (señal) para eliminar los efectos del campo gravitatorio. Además, añade las nuevas mediciones del sensor de aceleración (realizando los cálculos que sean necesarios) para que puedan ser utilizados por el algoritmo de detección de picos.

```
private void peakDetection()
```

Implementa el algoritmo de detección de picos[3], utilizando toda la información procesada en la función anterior. A medida que detecta nuevos pasos, los suma al atributo *mStepCounter*.

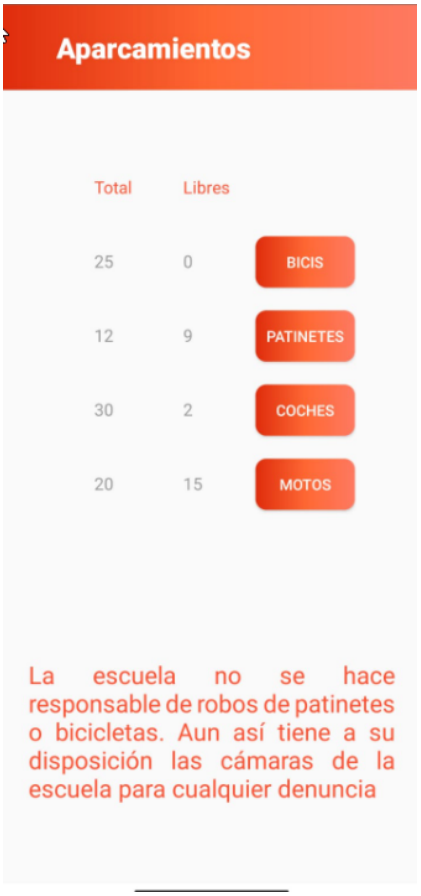
[3] Dicho algoritmo se desarrolla en *A Step Counter Service for Java-Enabled Devices Using a Built-In Accelerometer, Mladenov et al. ,* disponible en https://dl.acm.org/doi/10.1145/1554233.1554235

4.5 Clases asociadas al parking

4.5.1 ActivityParking

Descripción

Esta clase está asociada a la actividad Parking. Muestra una tabla con las plazas totales y disponibles de cada uno de los parkings (bicicletas, patinetes, coches y motos). Desde esta actividad se puede ir a la actividad Foto Parking la cual muestra una foto del parking seleccionado y si ese parking tiene plazas libres y cuantas.



Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Se usa cuando se crea la actividad, y este esta por defecto de android studio.

```
public void goBicis(View view)
public void goPatinetes(View view)
public void goCoches(View view)
public void goMotos(View view)
```

Estos 4 métodos son usados por los cuatro botones que contiene la interfaz de la actividad.
Todos ellos lanzan la actividad Foto Parking. Antes de lanzar dicha actividad, meten en el *intent*, una letra (B, P, C, M) para saber qué tipo de parking es el seleccionado. Esta letra es mapeada en el intent con el string “opcion”

4.5.2 ActivityTouchParking

Descripción

Esta actividad está asociada a la actividad Touch Parking. Esta solo muestra una indicación en texto para que se dibuje la letra inicial del parking que se quiera visualizar. Conforme se va dibujando la letra se ve en pantalla el dibujo en color amarillo.

Si la letra que se dibuja coincide con una de las guardadas en el archivo *gestures.txt* se lanza la actividad Foto Parking con la foto que corresponda dependiendo de la letra dibujada.

El dibujo se debe de hacer sin levantar la mano hasta que se acabe de dibujar la letra

Atributos

```
private Context context;
```

Almacena el contexto de la vista para pasarlo al intent de la actividad lanzada desde dentro del listener de los gestos.

```
GestureLibrary gesturelib
```

Almacena los gestos que serán comparados con el dibujo que hace el usuario

```
GestureOverlayView gestureview
```

Almacena la vista en la que se dibuja la letra

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Se lanza cuando se crea la actividad. Además de asignar el layout de la actividad, inicializa los atributos y le asigna un listener al `GestureOverlayView` para que cuando se dibuje la letra la compare y lance la actividad si coincide con una de las guardadas.

En el momento de asignar el listener, se sobreescribe el método `onGesturePerformed`

y en este se reconoce el gesto. Si tiene una puntuación mayor que 4, se lanza la actividad Foto Parking. Al igual que se hizo en la actividad Parking se le pasa en el intent una letra (B, P, C, M) dependiendo de la letra dibujada para que se muestre la foto al parking correspondiente.

Cada gesto está guardado con un nombre que empieza por la inicial en mayúscula y seguido de números. Esta letra inicial es la que indica a que parquing se refiere.

4.5.3 ActivityFotoParking

Descripción

Esta clase está asociada a la actividad Foto Parking. En ella se muestra una foto del parking. Se supone que esta foto sería una foto actualizada cada poco tiempo o un video en directo de una cámara del parking. También se muestra un texto indicando las plazas disponibles en ese momento, el cual estaría en verde o rojo dependiendo de si hay o no plazas disponibles.

Atributos

```
private TextView titulo, texto;
```

Almacenan la vista del textView del título de la actividad y del texto que muestra las plazas disponibles para ser modificados

```
private ImageView imgParking;
```

Almacena la vista de la imageView de la foto del parking

```
private int drawableResourceId;
```

Almacena el id del recurso drawable que se pondrá en el imageView

Métodos

```
protected void onCreate(Bundle savedInstanceState)
```

Además de asignar el layout a la actividad, inicializa los atributos de vista, obtiene la letra del parking desde el intent haciendo uso del string “opcion”.

Dependiendo de la letra que sea, asigna al imageView la foto correspondiente, obtenida desde la carpeta drawable. Se cambia el título y el texto de los textView *titulo* y *texto* y se le pone el color a verde o rojo al texto.

4.5.4 ShakeDetector

Descripción

Detecta si el teléfono está siendo agitado. Para ello comprueba si el 75% de las muestras están acelerando en los pasados 0.5s. Si es así está siendo agitado o está en caída libre desde alrededor de 1.84m

Atributos

```
public static final int SENSITIVITY_LIGHT = 11;
```

Almacena una sensibilidad baja para el sensor.

```
public static final int SENSITIVITY_MEDIUM = 13;
```

Almacena una sensibilidad media para el sensor.

```
public static final int SENSITIVITY_HARD = 15;
```

Almacena una sensibilidad alta para el sensor.

```
private static final int DEFAULT_ACCELERATION_THRESHOLD =
SENSITIVITY_MEDIUM;
```

Umbral de sensibilidad del sensor. Por defecto en medio

Métodos

```
public boolean start(SensorManager sensorManager)
```

Pone el sensor a escuchar. Devuelve verdadero si el teléfono soporta esta funcionalidad, es decir, si tiene o puede usar acelerómetro.

```
public void stop()
```

Para el sensor de forma segura

4.6 Clases asociadas a la base de datos SQLite

4.6.1 IndicacionesContract

Descripción

Esta clase representa el esquema de la base de datos, conteniendo la definición de todas las tablas de la base de datos.

En este caso, la base de datos solo contiene una tabla, representada en la siguiente clase abstracta:

```
public static abstract class IndicacionEntry implements BaseColumns
```

Esta clase implementa la interfaz *BaseColumns*, lo cual agrega una columna extra oculta, recomendado por los desarrolladores de SQLite para un mejor funcionamiento.

En su interior, la clase almacena el nombre de todos los atributos de la tabla (que se pueden ver en la clase *Indicacion*) y el nombre de la tabla:

```
public static final String TABLE_NAME ="indicaciones"
```

4.6.2 Indicación

Descripción

Esta clase representa una entidad de la base de datos (en este caso, la única que existe).

Atributos

```
private int orden
```

Orden de la indicación dentro de la ruta a la que está asociada

```
private String id
```

Número de identificación único

```
private String ruta
```

Nombre de la ruta a la cual está asociada a la indicación

```
private String imagen
```

URL que indica donde encontrar la imagen de la indicación

```
private String textoIndicacion
```

Texto que explica la acción que debe tomar el usuario (“Gira a la derecha”, “Continúa recto”,etc)

```
private int pasos
```

Número de pasos que se debe mostrar en pantalla una indicación antes de pasar a la siguiente.

Métodos

Como esta clase es principalmente un contenedor de datos, implementa un constructor y getters para todos los atributos (no incluye setters, pues en principio es una base de datos de solo lectura).

```
public ContentValues toContentValues()
```

Este método facilita la inserción de datos, convirtiendo los datos de entrada al formato que deben tener para que se ajusten al esquema de la base de datos.

4.6.3 IndicacionesDbHelper

Descripción

Clase que extiende a *SQLiteOpenHelper*, la cual se trata de una clase abstracta que provee los mecanismos básicos para la relación entre la aplicación Android y la información contenida en la base de datos.

Atributos

```
public static final int DATABASE_VERSION
```

Versión de la base de datos. Toma un valor mayor que 1, y cuando se modifica, se actualizan los datos (si se cambia a un valor mayor) o se reestablece una versión anterior (si el número es menor).

```
public static final String DATABASE_NAME = "BaseDatos.db"
```

Nombre de la base de datos.

Métodos

```
public IndicacionesDbHelper(Context context)
```

Constructor de la clase. Invoca al constructor de la clase padre.

```
public void onCreate(SQLiteDatabase sqLiteDatabase)
```

Este método es llamado automáticamente cuando se crea una instancia de la clase. En su interior se ejecuta la creación de la tabla *Indicaciones* y la carga de datos (realizada invocando la función *cargarDatos( )*).

```
public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion)
```

Este método es invocado cuando se cambia la versión de la base de datos a una mayor. Elimina las tablas almacenadas y las vuelve a crear.

```
public long nuevaIndicacion(SQLiteDatabase db, Indicacion
indicacion)
```

Este método facilita la inserción de una indicación en la base de datos, tomando los datos y convirtiéndolos al formato adecuado para que sea aceptado por SQLite.

```
private void cargarDatos(SQLiteDatabase db)
```

Realiza la inserción en la base de datos de todas las indicaciones asociadas a las diferentes rutas.

```
public ArrayList<Indicacion> obtenerRuta(String nombreRuta)
```

Devuelve las indicaciones asociadas a una determinada ruta, utilizando la consulta SQL

```
SELECT * FROM indicaciones WHERE ruta = '"+nombreRuta +"' ORDER BY orden
```

5. Bibliotecas externas

https://github.com/square/seismic

https://developer.android.com/training/gestures/multi

https://github.com/hannesa2/panoramagl

https://developers.google.com/ml-kit/vision/text-recognition

https://github.com/jjoe64/GraphView

https://github.com/isibord/StepTrackerAndroid

6. Bibliografía

https://github.com/isibord/StepTrackerAndroid

https://github.com/jjoe64/GraphView

https://www.geeksforgeeks.org/how-to-build-a-step-counting-application-in-android-studio/

https://developer.android.com

https://www.develou.com/android-sqlite-bases-de-datos/

https://pradogrado2122.ugr.es/pluginfile.php/93297/mod_resource/content/3/Multitouch.pdf

https://www.tutorialspoint.com/how-to-disable-landscape-mode-in-android