ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

Ingeniería Informática

Título del proyecto:

Sistema de etiquetado en Android para la valoración de la gravedad de los accidentes de tráfico

Gabriel Rocamador Murillo
Jesús Villadangos Alonso
Pamplona, Septiembre 2012

Índice

Índice de imágenes

1. Introducción

Desde la aparición del *Iphone* en el 2007 se ha producido un crecimiento exponencial del mercado de los denominados *smartphones*, y, con éste, también del software que corre en forma de aplicación por estos pequeños ordenadores. A lo largo de estas páginas se explicará el desarrollo de una aplicación para el sistema operativo Android (junto al iOs de Apple, uno de los sistemas operativos para dispositivos móviles más popular).

El contexto de la aplicación son los accidentes de tráfico. Existen diferentes estudios sobre escenas de accidentes de tráfico que relacionan factores como el lugar del impacto o el hundimiento del impacto con las lesiones producidas en las victimas. Una manera de medir estos factores es mediante el IDE (Índice de deformación estructural) **[1]**. De modo que conocer la escena de un accidente puede ayudar a atender más eficientemente a sus victimas.

Se desarrolló en 2010 una aplicación de escritorio que calcula el índice de deformación estructural de un accidente concreto **[2]**, a diferencia de esta, la aplicación desarrollada no está específicamente orientada a este fin, sino que trata más a fondo la recogida de datos de accidentes de un modo directo a través de dispositivos móviles. En definitiva, el objetivo de esta aplicación es clasificar y catalogar los puntos críticos de siniestros automovilísticos con el fin de ayudar en la toma de decisiones de éstos. Se trata de un prototipo de recogida de datos de accidentes que sirva de apoyo para un proyecto más grande, bien para la determinación de IDE antes comentado, o bien para otros estudios.

El usuario final (un usuario especializado: bien un agente de tráfico, bien un sanitario, un científico etc.) deberá ser capaz de reproducir en una escena un accidente concreto.

La forma elegida de proporcionar dicha funcionalidad fue mediante puntos, localizados sobre una silueta que representa al vehículo siniestrado.

Permitiendo al usuario añadir la siguiente información a cada punto:

Gravedad: dividida en tres categorías; grave, medio y leve.

Imagen: se permite añadir una imagen por punto etiquetado, la imagen corresponderá a una fotografía real tomada en dicho punto del vehículo.

Las imágenes se podrán tomar tanto de la cámara de fotos del dispositivo, lo cual sería muy útil si el usuario es un agente de atestados por ejemplo, como de la galería de imágenes.

En la figura 1 podemos ver un ejemplo de una escena de accidente con tres puntos etiquetados con colores (que indican la gravedad) y dos de ellos acompañados por el icono de una fotografía (que indica que dichos puntos están acompañados de una imagen).

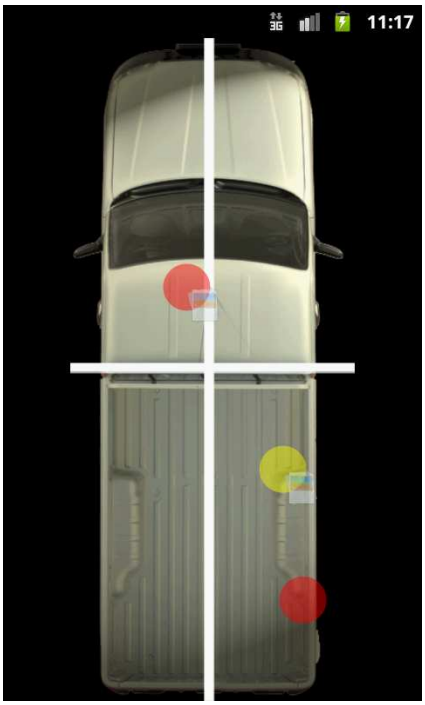


Fig. 1 Ejemplo de escena

Además para cada siniestro se almacenarán: un nombre que lo identifique, el tipo de vehiculo accidentado y será opcional añadir una descripción del siniestro.

Todos estos datos recogidos para cada siniestro, deberán almacenarse de manera persistente.

El hecho de tratarse de una aplicación móvil implica diferentes cuestiones. Obviamente hablamos de una aplicación táctil, y por tanto deberá ser una aplicación sencilla (que no simple) e intuitiva. Además habrá que tener en cuenta los recursos utilizados para no saturar el dispositivo.

2. Desarrollo en Android

La aplicación desarrollada, está programada para el sistema operativo Android, propio de los dispositivos móviles como móviles o tablets, y que está basado en Linux.

La manera más habitual de desarrollar aplicaciones en Android, y la manera en que esta aplicación ha sido desarrollada es mediante el lenguaje de programación Java y un API proporcionado por Android conocido como *Android Software Development Kit* (SDK) **[3]**.

No es necesario, pero si conveniente, conocer los detalles de la arquitectura a más bajo nivel, por tanto se hará un breve resumen de la arquitectura del sistema de Android, para después comentar tres conceptos básicos del desarrollo de aplicaciones para este sistema operativo.

2.1 Arquitectura de Android

La arquitectura del sistema de Android se resume en el siguiente cuadro:

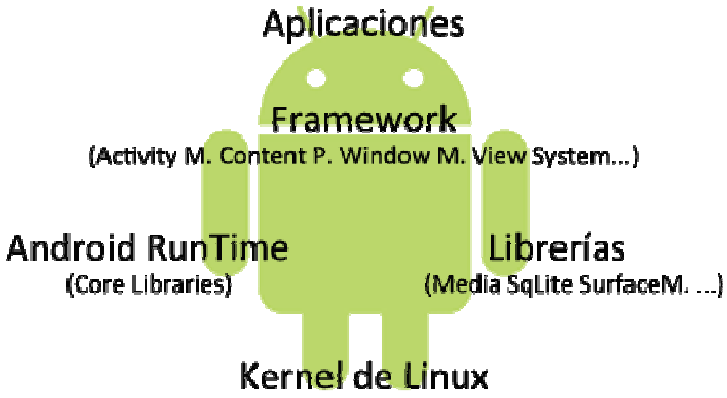


Fig. 2 Arquitectura del sistema

A continuación se explica brevemente los componentes vistos en el cuadro:

Kernel de Linux: Hace de interfaz o capa de abstracción entre el software y el hardware, proporcionándole además servicios base (de memoria, seguridad, gestión de procesos...).

Librerías: Android incluye bibliotecas C/C++ que usan diferentes componentes del sistema. Algún ejemplo de estas son *SGL* (motor de gráficos 2D), *SqLite* (para crear pequeñas bases de datos relacionales) etc.

Android RunTime: Es el software que hace correr las aplicaciones. Para ello cuenta con *Core libreries* que son un conjunto de librerías de bajo nivel.

Además utiliza el Kernel de Linux para gestionar las aplicaciones, de modo que una aplicación, equivale a un proceso.

Framework: Es la plataforma creada para los desarrolladores, a los que se proporciona acceso a los mismos APIS que las aplicaciones base (las que vienen por defecto). Este Framework permite la reutilización de componentes.

Aplicaciones: Las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros. Estás usan como ya se ha dicho el mismo Framework que el que usan los desarrolladores de nuevas aplicaciones.

2.2 Conceptos básicos

A continuación se van a analizar tres conceptos básicos para comprender en que consiste una aplicación de Android, son los siguientes: Android Manifest , Actividad e Intent:

Actividad (*Activity*): es una de las cuatro clases fundamentales para una aplicación y la más importante. Junto a los *Services* (para operaciones en segundo plano), *Content Providers* (acceso a estructuras de datos) y *Broadcast Receivers* (receptor de eventos) forman los cuatro componentes fundamentales. **[4]**

Una actividad es el foco de lo que el usuario está haciendo, generalmente se le asigna la UI (*User Interface*) usando la función *setContentView(int)*. Una actividad pues, suele representar una ventana o pantalla de la aplicación, aunque también pueden ser usadas de otra manera como ventanas flotantes o embebidas en otra actividad.

Las actividades tienen un ciclo de vida (figura 3), y deben ser declaradas en el archivo *AndroidManifest.xml*.

Android Manifest: Es un archivo XML que deben tener todas las aplicaciones Android en su directorio raíz. Contiene información esencial de la aplicación:

- El nombre del paquete o identificador de la aplicación.
- Describe las clases que implementan los componentes fundamentales, usados en la aplicación y sus funciones (por ejemplo si la actividad es la principal).
- Declara los permisos que necesita la aplicación y los que necesita otros para interactuar con esta.
- La versión mínima del API necesaria para correr la aplicación.
- Las clases *Instrumentation*, que sirven para monitorear.
- Las librerías que no se incluyen automáticamente son listadas aquí.

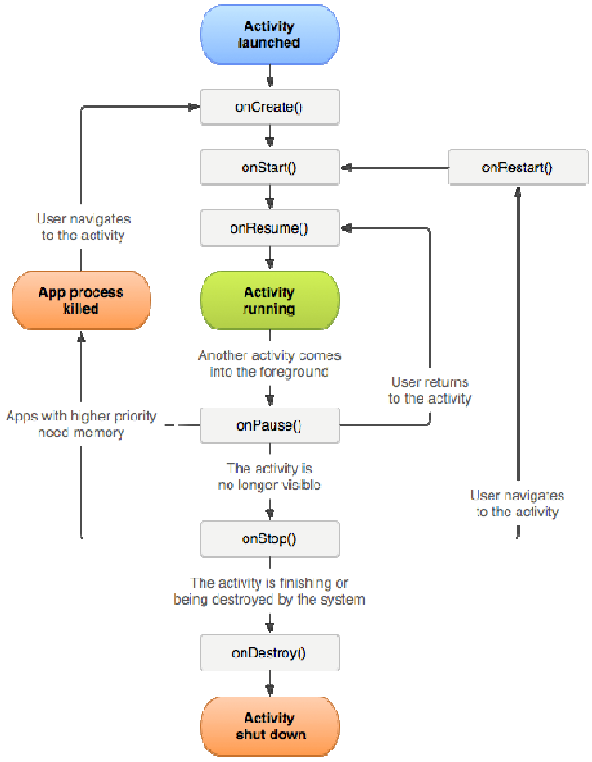


Fig. 3 Ciclo de vida de una Actividad

Intent: Un *Intent* es una descripción abstracta de una operación a realizar, en forma de mensajes asíncronos los *Intents* permiten a los componentes de una aplicación solicitar funcionalidad al sistema y comunicarse entre ellos.

Podemos hacer que nuestras actividades respondan a *Intents* (como cambios de estado en el hardware, notificaciones de datos entrantes...). También podemos crear *Intents* que lancen actividades u otros componentes, bien explícitamente (nombrando el componente) o implícitas (nombrando la acción, el sistema operativo elegirá el componente adecuado o listará los disponibles).

Existen muchos más conceptos y clases fundamentales para poder comprender la estructura de una aplicación Android (como por ejemplo la clase *View*, los Layout…) que bien podrían abarcar una memoria completa, sin embargo que aquí se pretende esbozar en que consiste la arquitectura básica de una aplicación android.

3. Arquitectura de la aplicación

A continuación veremos la arquitectura específica de la aplicación a desarrollar. No se trata de una arquitectura compleja puesto que es una aplicación móvil sin conexión a servidores externos ni grandes bases de datos. Sí que existirá el acceso a datos almacenados en la tarjeta de memoria del dispositivo (*SDCard*) en el que se instale la aplicación.

En la figura 4 podemos identificar las diferentes pantallas de la aplicación y cómo estas acceden a los datos bien para leer (flechas que señalan hacía éstos) o para escribir (flechas inversas).

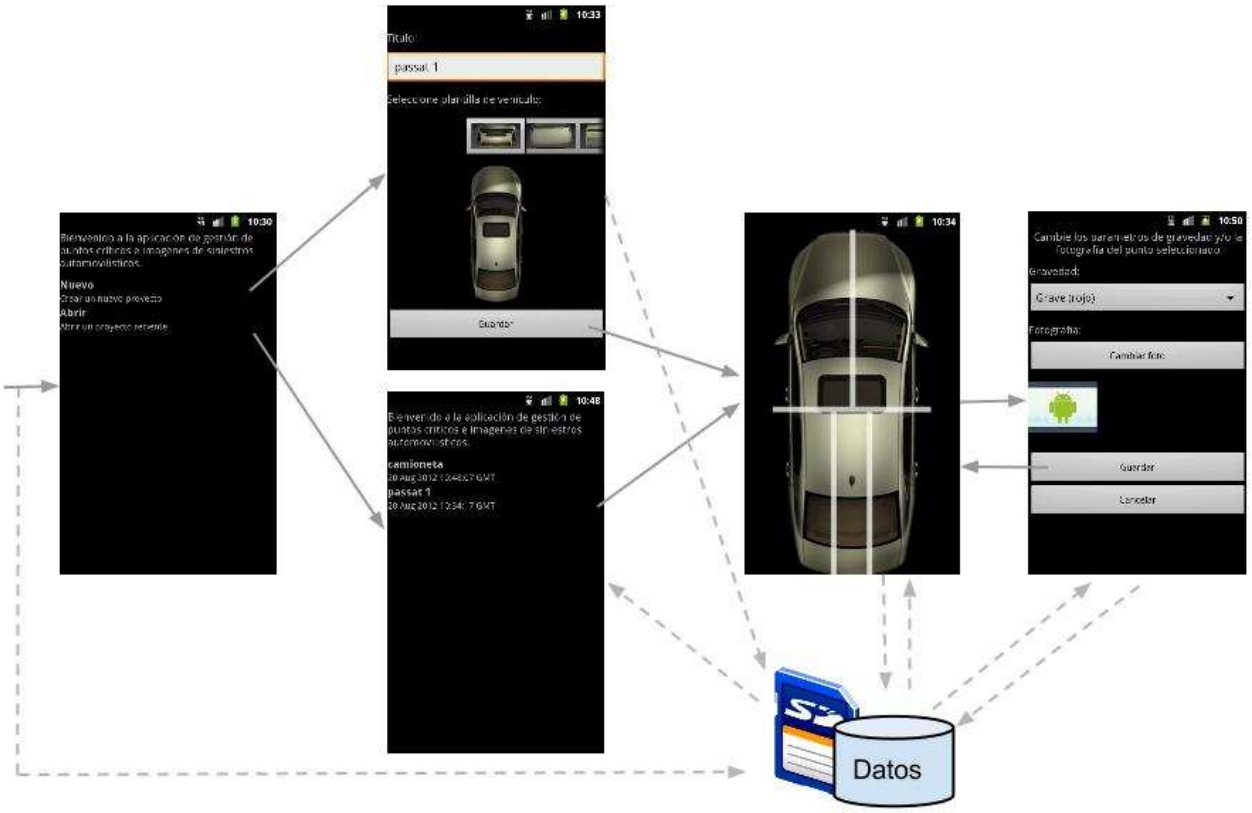


Fig. 4 Arquitectura de la aplicación

Podemos ver que antes de la primera pantalla ya existe un acceso a datos, esto, como se explicará mejor más adelante, es debido a que creamos una carpeta para nuestra aplicación en la *SDCard* donde guardaremos toda la información que recojamos de cada escena o accidente. La primera pantalla se divide en dos, dependiendo de si decidimos crear un nuevo proyecto (o escena) o abrir uno existente. Si abrimos lógicamente haremos un acceso de lectura a dato, y al crear uno de escritura.

Las otras dos pantallas leen y escriben indiscriminadamente, debido a que ambas contienen prácticamente toda la funcionalidad de la aplicación. Esta funcionalidad se describirá detalladamente en el siguiente tema

4. Requisitos

El objetivo principal del proyecto es permitir al usuario la obtención de datos de accidentes de tráfico a través de un dispositivo móvil Android. A lo largo de éste capitulo detallaremos cuales son los requisitos dados, que tienen como propósito cumplir el objetivo final. Existen muchas maneras de lograr dicho objetivo, por lo que antes de comenzar a desarrollar se decidió la manera en la que se daría solución al problema. Se intentó que resultara una aplicación sencilla y atractiva de fácil manejo.

4.1 Descripción del problema

Creación de escenas:

El usuario podrá crear una nueva escena sobre la que recoger datos de un accidente, le dará un nombre único que la identifique. El usuario deberá elegir la plantilla que más se adecue al tipo de vehiculo siniestrado. Las escenas elegidas para el programa son las de la figura 5 Además podrá añadir una descripción a la nueva escena.

Fig. 5 Siluetas

Creación de puntos

El usuario podrá añadir un punto dentro de los límites de la silueta, lo añadirá tocando sobre la silueta y el punto se creará y dibujará en las coordenadas presionadas. El punto dibujado consistirá en un círculo del diámetro aproximado de la huella dactilar de un dedo, puesto que en un dispositivo táctil la precisión no es muy elevada.

Para cada punto tocado deberá iluminarse la zona del coche correspondiente (piloto, copiloto, asiento izquierdo…) y almacenarse en la información del punto, esto servirá para posteriores estudios.

El usuario podrá añadir al punto creado la gravedad del golpe en dicho punto. Esto debe hacerse de manera intuitiva desde la misma pantalla.

Se optó por hacer que el sistema mostrará la imagen de un pequeño semáforo al lado del punto y que al presionar cada color del semáforo (rojo, ámbar y verde) el punto cambiara de color, indicando cada color un nivel de gravedad (grave, medio y leve respectivamente) pudiendo también no definir gravedad al punto.

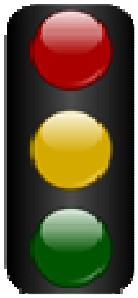


Fig. 6 Semáforo "para la gravedad"

El usuario podrá añadir al punto creado una imagen, fotografiarla en ese momento u obtenerla de la galería. Una vez añadida la imagen se le añadirá al punto un icono de una fotografía, el usuario tocando el icono podrá ver la imagen añadida al punto. El usuario podrá cambiar la imagen si lo desea.

Fig. 7 Icono "imagen añadida"

El usuario podrá guardar el punto creado o desecharlo. Cuando el punto se haya guardado quedará dibujado en la silueta con su color correspondiente y se indicará con el icono anterior en pequeño si el punto posee imagen.

Edición de puntos:

El usuario deberá ver en todo momento de forma gráfica y visual todos los puntos que ha creado. El usuario podrá seleccionar un punto tocándolo. Una vez seleccionado si el punto posee imagen el usuario podrá verla tocando en el icono. El usuario podrá editar el punto (gravedad e imagen), borrarlo o no hacer nada.

Edición de escenas:

El usuario podrá abrir escenas o accidentes de tráfico previamente creados y visualizarlos. También podrá modificarlos: añadiendo, editando o borrando puntos de éste; añadiéndoles descripción y cambiando la fecha.

El usuario podrá borrar la escena que desee.

4.2 Diagrama de casos de uso

Analizando la descripción del problema podemos definir diagrama de casos de uso, a nivel general como el de la figura 8. Se podría seguir refinando creando más casos de uso sin embargo lo que se hará es, en el siguiente punto, definir todos los requisitos encontrados en lenguaje natural.

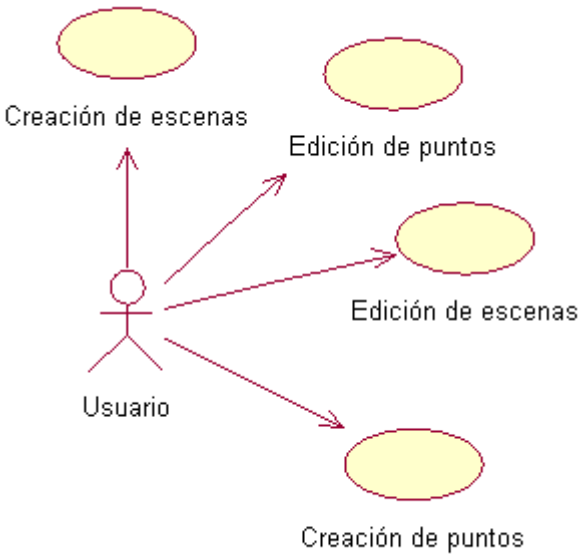


Fig. 8 Diagrama de casos de uso

4.2 Requisitos funcionales

A continuación se desglosan los requisitos funcionales obtenidos de la descripción, estos son más específicos y detallan lo que es necesario implementar y que proporciona funcionalidad al proyecto.

Creación de escenas:

- El sistema debe comprobar que el nombre dado es válido. En caso de que el nombre este repetido o en blanco el programa debe informar al usuario.

- El sistema deberá almacenar la información de la silueta en el proyecto, junto al nombre y la fecha de creación de éste (con el fin de listarlos posteriormente por fecha).

- El sistema debe mostrar en una pantalla la silueta escogida para que el usuario trabaje con ella.

Creación de puntos:

- El sistema deberá dibujar sobre la silueta el punto tocado por el usuario en forma de círculo si está en los límites de la silueta.

- El sistema no dibujará nada si se ha tocado fuera de los límites de la silueta.

- El sistema debe guardar cada punto con sus coordenadas *X* e *Y*, y un circulo concéntrico del mismo tamaño para todos los puntos.

- Los círculos asociados a cada punto no deben unirse y deben tener una separación entre ellos mínima, con el fin de evitar confusiones.

- El sistema no podrá permitir crear más puntos cuando un punto se está editando, para evitar la confusión.

- El sistema mostrará la zona correspondiente al punto iluminada y almacenará la zona en el punto creado. Por tanto cada silueta habrá que dividirla en zonas, y crear el efecto óptico de la iluminación al pulsar ima de estas zonas. Como podemos ver en la figura 5, para la silueta de vehiculo normal, cada vez que creemos un punto en una de las zonas delimitadas por las franjas blancas la imagen de fondo cambiará en función de la zona. Esto se hará para todas las siluetas.

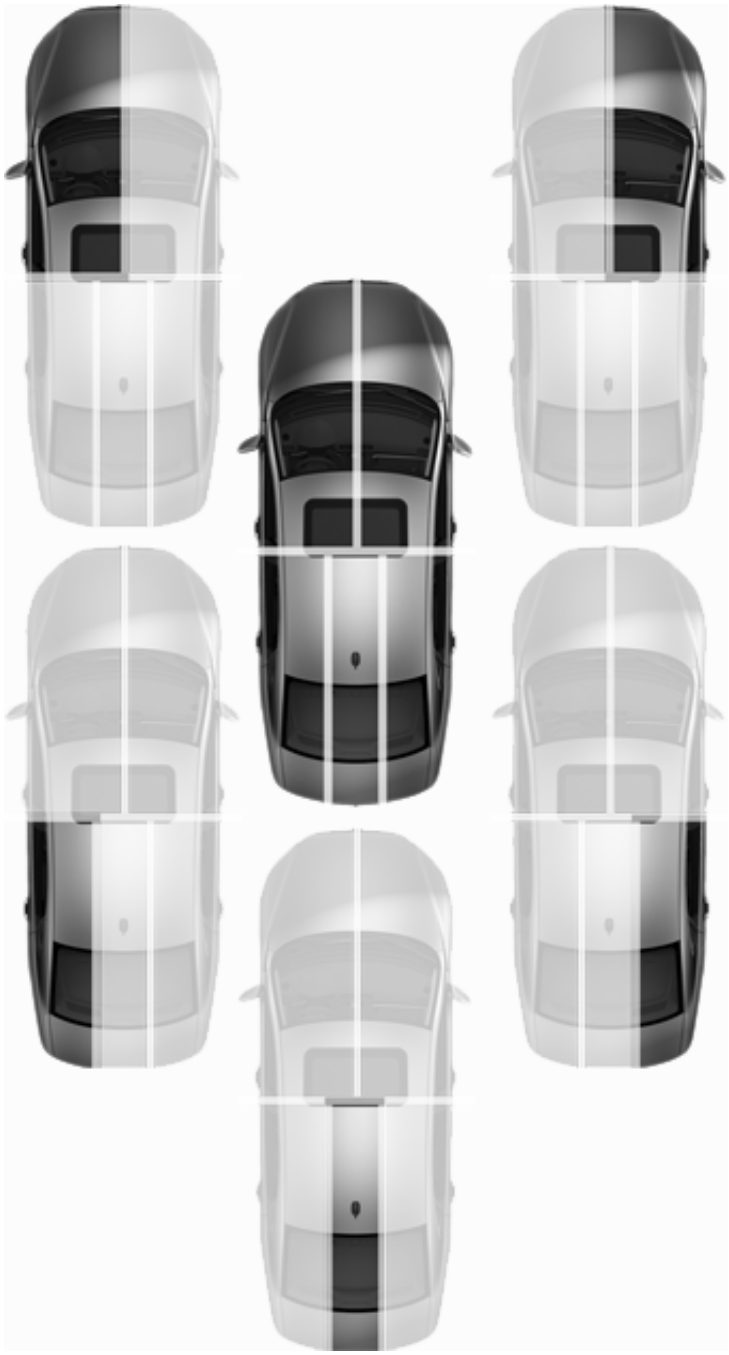


Fig. 9 Zonas de vehículo de la silueta 1 al tocar cada zona

- El sistema mostrará un semáforo sin colores iluminados, de modo que cuando se presione uno de ellos cambie de estado a ese color. Y así siempre que se presione un color menos cuando ese color ya estuviera seleccionado, en ese caso el semáforo volvería a la posición original.

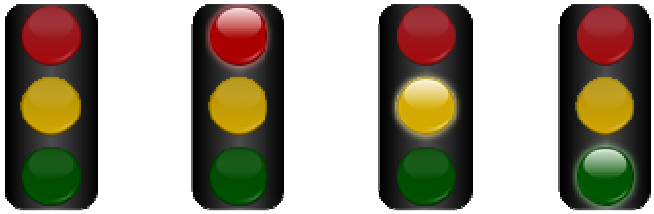


Fig. 10 Semáforo, tras selección de colores

- El sistema debe permitir obtener fotos tanto de la galería como de la cámara. Si la imagen ha sido obtenida con éxito el sistema mostrará al lado del punto el icono correspondiente (fig. 7).

- Al pulsar en el icono el sistema mostrará en una nueva ventana o dialogo la imagen asociada a ese punto.

- El sistema almacenará el punto únicamente si el usuario lo ha guardado.

Edición de puntos:

- El sistema debe almacenar todos los puntos y permitir recogerlos para modificarlos o borrarlos.

- El sistema debe ser capaz de dibujar todos los puntos guardados para una escena dada.

- El sistema deberá ser capaz de identificar cuando un usuario a pulsado una zona si está tratando de seleccionar un punto, y recuperar dicho punto.

Edición de escenas:

- El sistema deberá ser capaz de listar las escenas existentes por nombre.

- El sistema debe ser capaz de borrar una escena. Debe confirmar esta acción con el usuario a través de un dialogo como el siguiente:

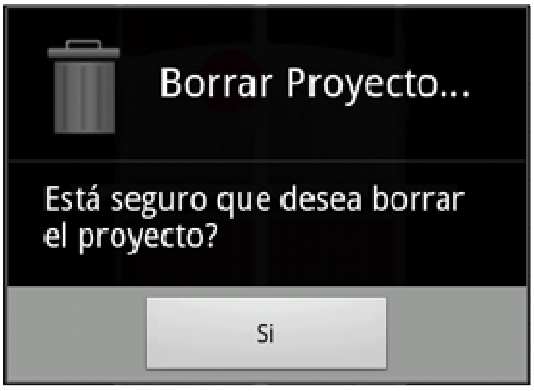


Fig. 11 Dialogo borrar proyecto

4.3 Requisitos no funcionales

Los requisitos no funcionales son aquellos que no especifican un determinado comportamiento o función, sino una característica requerida del sistema, del proceso de desarrollo, del servicio prestado o de cualquier otro aspecto del desarrollo, que señala una restricción del mismo. A continuación los más significativos:

- Las siluetas deberán poder añadirse de manera sencilla por un programador, en aras de la escalabilidad del proyecto.

- La aplicación debe poder ejecutarse en cualquier dispositivo móvil o tableta. Todas las coordenadas deben almacenarse en proporción a la pantalla y silueta y no en pixeles, para hacerlo independiente de la resolución de pantalla. Para ello debemos obtener la anchura y la altura de la imagen, antes de que Android la dibuje en la vista principal del siguiente modo:

```
vto.addOnPreDrawListener(new ViewTreeObserver.OnPreDrawListener() {
   public boolean onPreDraw() {

                    altura = imagenFondo.getMeasuredHeight();
                    ancho = imagenFondo.getMeasuredWidth();

                    return true;
               }
});
```

- En el momento que se decide guardar el punto creado el sistema deberá hacer ese punto persistente. Con el fin de evitar perdida de datos si el usuario u otro evento cierra la aplicación en un momento cualquiera.

- El sistema no debe permitir que la pantalla sea girada puesto que el tamaño de la silueta cambiaría no siendo fiel a la realidad. Esto se soluciona con una simple orden en la Activity:

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT)
```

- El tamaño de las imágenes deberá ser adecuado para que el dispositivo no cargue excesiva memoria con la aplicación.

- El sistema deberá controlar cada vez que el usuario presione el botón *back* de su dispositivo.

5. Análisis

A continuación estudiaremos como llevar a cabo la funcionalidad descrita en el capítulo interior. Debemos sacar de los requisitos, las entidades que van a diseñarse e implementarse posteriormente. Además veremos que entidades deben persistir, es decir, es necesario almacenar y cual es la mejor manera de hacerlo.

5.1 Identificando Clases

Vamos a tratar de identificar las clases principales que deberemos de implementar. A la hora de implementar probablemente necesitaremos definir algunas clases más.

Observando los requisitos podemos diferenciar dos clases principales que tendremos que implementar:

Escena (Proyecto): cada proyecto o escena será un objeto, con atributos, nombre, fecha, descripción y un conjunto de Puntos.

Punto: un objeto punto posee siempre sus coordenadas, que le identifican junto al nombre proyecto al que pertenece (no podremos crear dos puntos en el mismo proyecto en las mismas coordenadas), además tiene campos opcionales como la gravedad (puede estar no definida) y la imagen.

Además existen otras clases más difíciles de identificar. Por ejemplo:

Semaforo: Esta clase nos vendría muy bien, ya que cada vez que se crea un punto bastará con instanciarla y que el objeto se encargue de la funcionalidad de cambiar de color al ser tocardo. Para poder hacer esto la clase debe heredar de la clase *View* de *Android.Object*. (El trabajo con las vistas lo explicaremos más adelante en la implementación). El color marcado será un atributo de esta clase.

Imagen: También identificamos, al igual que con el semáforo, la clase Imagen, que incluye el icono de la imagen y la dirección, y contendrá la funcionalidad de tocar el icono.

Por último debemos identificar los interfaces a crear para la interacción entre el usuario y el programa, al tratarse de un programa Android estos se implementan como clases *Activity*, que leen un archivo Layout de tipo XML. Son los siguientes:

Formulario de Inicio: Donde podemos elegir entre crear un nuevo proyecto (o escena) o abrir uno existente. En este mismo formulario mostraremos los proyectos existentes cuando el usuario trate de abrir uno.

Formulario de Nuevo Proyecto: En este Formulario es donde elegimos el nombre del nuevo proyecto y la silueta que mejor se adecue al siniestro a analizar.

Formulario principal: Este formulario será el más complejo puesto que aquí vamos a desarrollar toda la aplicación. A él llegaremos desde un nuevo proyecto o abriendo uno existente y en él haremos la recogida de datos: Creación de puntos, edición de puntos, añadir información al proyecto, fotografías, borrar el proyecto…

5.2 Entidades persistentes

No todas las clases identificadas han de ser persistentes. Tan sólo debemos almacenar algunas de ellas, esto dependerá de los requisitos. En este caso se dice que debemos almacenar los puntos de cada proyecto para poder mostrarlos cuando se abre un proyecto. Para poder listar los proyectos existentes también será necesario almacenarlos. Por tanto, nuestras entidades serán la clase Punto y la clase Proyecto.

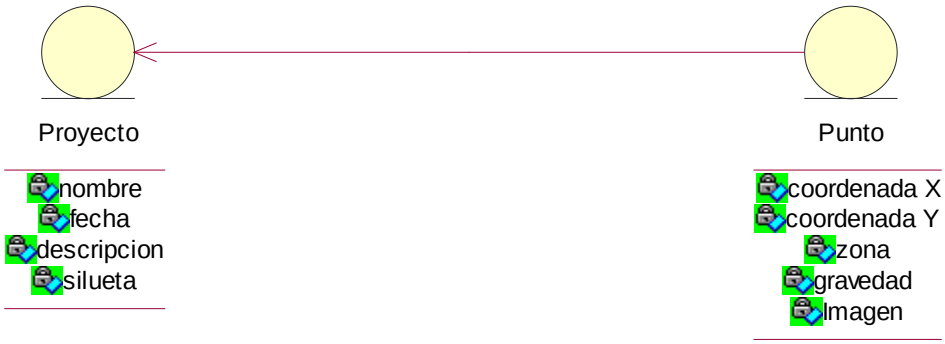


Fig. 12 Entidades persistentes

En la figura 12 hemos definido los atributos para cada clase y además en la relación de asociación hemos indicado que los puntos pertenecen a un proyecto dado. De modo que todo punto tendrá un proyecto y los proyectos podrán contener puntos.

5.3 Almacenamiento de datos

Ahora que sabemos que datos vamos a guardar tenemos que pensar cómo hacerlo. Para decidir el modo de almacenamiento debemos tener en cuenta varios aspectos de nuestra aplicación. En primer lugar, nuestra aplicación va a almacenar, además de datos simples, imágenes. Estas imágenes deberemos guardarlas en una subcarpeta del directorio raíz de la *SDCard* del dispositivo Android.

Los datos a guardar son escasos, si construyéramos una base de datos tan sólo poseería 2 tablas, y a una de ellas, la de proyectos, sólo se accedería para abrir un proyecto, editarlo o borrarlo. Por tanto, la opción escogida para almacenar los datos es la siguiente:

- Al instalar por primera vez la aplicación se creará una carpeta nueva en nuestro directorio raíz, a la que llamaremos *CarMedia*.
- Para cada proyecto (escena) creado se creará una carpeta con el nombre del nuevo proyecto en la carpeta de la aplicación.
- Para cada proyecto (escena) creado se creará un archivo XML con el nombre del nuevo proyecto *.xml*, en la carpeta de la aplicación con los datos del proyecto (nombre, fecha, silueta...).
- Por cada punto creado, se añadirán todos los datos del punto al fichero XML del proyecto.
- Las imágenes sacadas con la cámara de fotos también se almacenarán en dicha carpeta. Las provenientes de la galería de imágenes no lo harán, con el fin de evitar redundancia.

Para poder crear carpetas y archivos desde nuestra aplicación a un dispositivo Android necesitamos el permiso del usuario al instalar la aplicación. Para solicitárselo debemos añadir en el *AndroidManifest* la siguiente línea:

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

A continuación vamos a explicar el fichero XML que se va generando en cada proyecto que creamos.

Supongamos que acabamos de crear un proyecto de nombre *p1* a fecha 21 de Agosto, he elegido la silueta 1, la que corresponde a un vehiculo normal. El XML que debe generarse es el siguiente:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<Proyecto Nombre="p1">
  <Nombre>p1</Nombre>
  <Fecha>21 Aug 2012 18:29:43 GMT</Fecha>
  <Silueta>
    <nombreSilueta>sample_1</nombreSilueta>
    <descripcion>Coche normal</descripcion>
  </Silueta>
  <Puntos />
</Proyecto>
```

El fichero se crea con 4 etiquetas hijo. El nombre, la fecha (que se crea automáticamente), la silueta, donde cada una de las tres siluetas tiene por nombre *sample_X* siendo *X* el número de la silueta, y por último una etiqueta vacía denominada *Puntos*.

Ahora supongamos que creamos un punto en el proyecto anterior. Lo definimos con gravedad color rojo, y decidimos no añadirle fotografía alguna. El fichero anterior se editaría de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?><Proyecto Nombre="p1">
  <Nombre>p1</Nombre>
  <Fecha>21 Aug 2012 18:29:43 GMT</Fecha>
  <Silueta>
      <nombreSilueta>sample_1</nombreSilueta>
      <descripcion>Coche normal</descripcion>
  </Silueta>
  <Puntos>
      <Punto>
            <Posicion>
                  <X>0.6346556</X>
                  <Y>0.33906883</Y>
            </Posicion>
            <Zona>
                  <NumZona>1</NumZona>
                  <Descripcion>Copiloto</Descripcion>
            </Zona>
            <Gravedad>1</Gravedad>
            <Foto/>
      </Punto>
   </Puntos>
</Proyecto>
```

Vemos que el punto se ha añadido como una etiqueta hijo de la etiqueta *Puntos*, éste a su vez contiene otras etiquetas; la posición se guarda como dos coordenadas *X* e *Y* entre 0 y 1, esto lo hemos decidido así para que sea lo más equivalente posible en cualquier resolución. Además de este modo es más fácil para trabajar a posteriori; por ejemplo, si *Y* vale *0.5* sabemos que el golpe se encuentra en la línea que parte por el punto medio el vehiculo.

La zona del golpe señalada es la del copiloto, y la gravedad del golpe es grave, ya que a cada gravedad le asignamos un entero (1 grave, 2 medio, 3 leve). Por último tenemos la etiqueta *Foto* que está vacío porque no hemos añadido ninguna foto en el punto. Si editamos el punto y le añadimos una fotografía de la cámara, el XML para la etiqueta *Foto* cambiaría de la siguiente manera:

```
<Foto>/mnt/sdcard/CarMedia/p1/1345626549726.jpg</Foto>
```

Se guarda la dirección de la imagen, como la acabamos de obtener de la cámara es necesario nombrar la foto. Para evitar que sea el usuario quien le ponga nombre se le pone por nombre la el tiempo del sistema en milisegundos.

Cuando queramos borrar un punto lo quitaremos del fichero XML, mientras que cuando queramos borrar un proyecto lo que haremos será borrar toda la carpeta del proyecto.

6. Diseño e Implementación

Una vez hemos analizado que clases debemos crear, y el almacenamiento de datos, describiremos cómo se ha llevado a cabo el diseño y la implementación. Lo primordial será conocer cual es la plataforma de desarrollo con lo que se ha contado para desarrollar la aplicación.

Después analizaremos las clases desarrolladas y veremos en el diagrama de clases las relaciones entre éstas. Por último, puede resultar interesante conocer algunos detalles técnicos sobre funcionalidades importantes desarrolladas.

Este capítulo será el más extenso. Se trata de una aplicación desarrollada por un único programador, con experiencia en desarrollo ágil, por lo que se dedicó menos tiempo al análisis.

6.1 Plataforma

Es posible programar aplicaciones para Android usando diversos lenguajes de programación, como Java, C++, HTML5... El elegido para el desarrollo fue Java, por conocerlo previamente, porque es el más extendido y el ideado originalmente para desarrollar aplicaciones Android, y porque con él podremos usar un API muy completo llamado Android SDK (*Software Development Kit*).

El entorno elegido para programar es Eclipse**[5]** que junto con el complemento ADT (*Android Development Tools plugin*) es el soporte oficial de la SDK. Eclipse está disponible para cualquiera de los sistemas operativos mayoritarios, en éste caso se usó Windows XP. La versión del API que se ha usado es la 8, lo que equivale a la versión de Android 2.2 para los dispositivos móviles, esto significa que la aplicación no podrá instalarse en versionas anteriores y sí en la propia y posteriores.

Cabe mencionar que el tratamiento de todas las imágenes se ha realizado mediante el programa *PhotoShop*.

6.2 Diagramas de clase

Esta sección es una de las más importantes, puesto que analizaremos todas las clases creadas, con sus atributos y sus funcionalidades más importantes. Las clases principales son las que identificamos en la parte del análisis, sin embargo, a lo largo del desarrollo se crearon clases de apoyo que explicaremos más adelante. También veremos como estas clases interactúan entre ellas, y por último se mostrará el diagrama de clases completo.

6.2.1 Clases

Comenzamos a definir las clases en el mismo orden en el que las identificamos:

Punto:

En la figura 13 podemos ver el diagrama de la clase Punto:

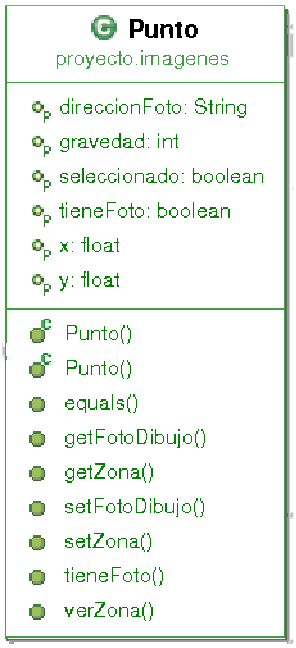


Fig. 13 Clase Punto

Podemos observar que tiene dos constructores. La explicación de esto es sencilla. Vamos a crear puntos de dos maneras diferentes. La primera cuando hayamos creado un punto nuevo, en ese caso al constructor se le pasará las coordenadas *X* e *Y* en píxeles, y también la anchura y altura para proporcionar las coordenadas. Sin embargo, como acabamos de crear el punto, aún no sabemos su gravedad ni si posee foto:

```
public Punto(Context context, float x, float y, int ancho, int
altura){
      ....
}
```

El otro modo de crear un punto es cuando lo acabamos de cargar del XML, en ese momento sabremos sus coordenadas ya proporcionadas, sabemos también su gravedad y si tiene foto:

```
public Punto(Context context, float x, float y, int gravedad,
                  int zonaVehiculo, boolean tieneFoto) {
       ....
}
```

Le pasamos una clase de tipo *Context*, para que el punto pueda trabajar con los recursos de la actividad desde la que se crea. Por ejemplo, para calcular la zona, necesita conocer las medidas de la silueta, que son un recurso.

Una función que vemos que está sobrescrita es Equals(); como ya habíamos indicado la igualdad entre dos puntos en un proyecto no es la igualdad entre sus coordenadas, sino que tomamos un margen para que los círculos que envuelven las coordenadas no se toquen.

Proyecto:

En la figura 14 podemos ver el diagrama de la clase Proyecto, a la hora de desarrollar decidí nombrarla *XmlProyecto* porque básicamente esta clase implementará todas las funciones de lectura y escritura en el XML: guardar, editar, cargar y borrar puntos, y guardar editar y borrar un proyecto.

Fig. 14 Clase Proyecto

Tras crear un objeto *XmlProyecto* tendremos dos opciones para llenar sus atributos: crear un nuevo proyecto, en cuyo caso llamaremos a la función *escribirProyecto(String nombre,String silueta);* o abrir un proyecto existente con la función *abrirXml(String direccion).*

La función *cargarPuntos()* devuelve un array de objetos de tipo Punto, correspondientes a todos los que han sido escritos en el XML con la función *addPunto(Punto p)*. De igual modo tenemos funciones para editar y eliminar puntos.

Semaforo

Vamos a ver rápidamente la clase *Semaforo* que hereda de la clase *View*. Al heredar de esta clase podemos luego añadirla a nuestra vista principal como una vista, y también podemos sobrescribir el evento *OnTouchEvent*, que saltará cuando alguien toque nuestra vista. Esto lo explicaremos en profundidad más adelante en el punto Detalles técnicos.

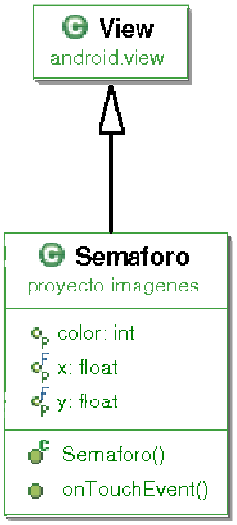


Fig. 15 Clase semáforo

FotoDibujo y DrawCanvasCircle:

Al igual que un objeto *Semaforo*, también el punto debe ser dibujado, y junto a él, cuando éste posee imagen, debe haber un ícono de fotografía que al tocar muestre la imagen. Pues bien, *DrawCanvasCircle* será el encargado de dibujar el punto mientras que *FotoDibujo* será el encargado de mostrar el icono de la imagen y mostrar la imagen cuando se pulse en él.

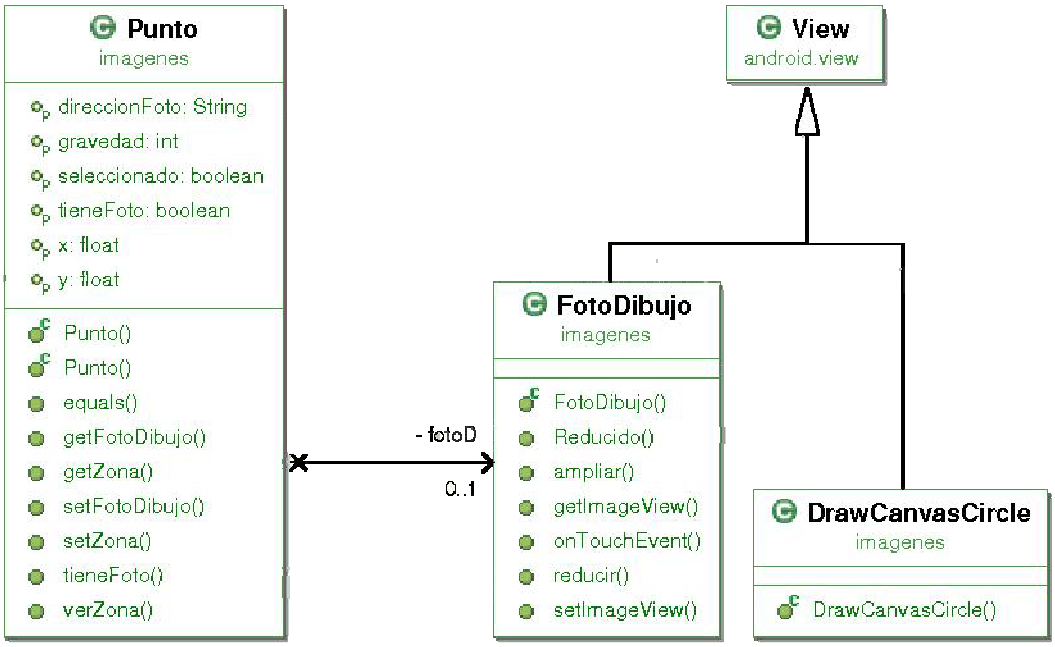


Fig. 16 FotoDibujo y otras clases

Las funciones de *FotoDibujo ampliar*() y *reducir*() sirven para aumentar el icono cuando el punto está seleccionado, o disminuirlo cuando lo deseleccionamos. Como podemos apreciar en la figura 17:

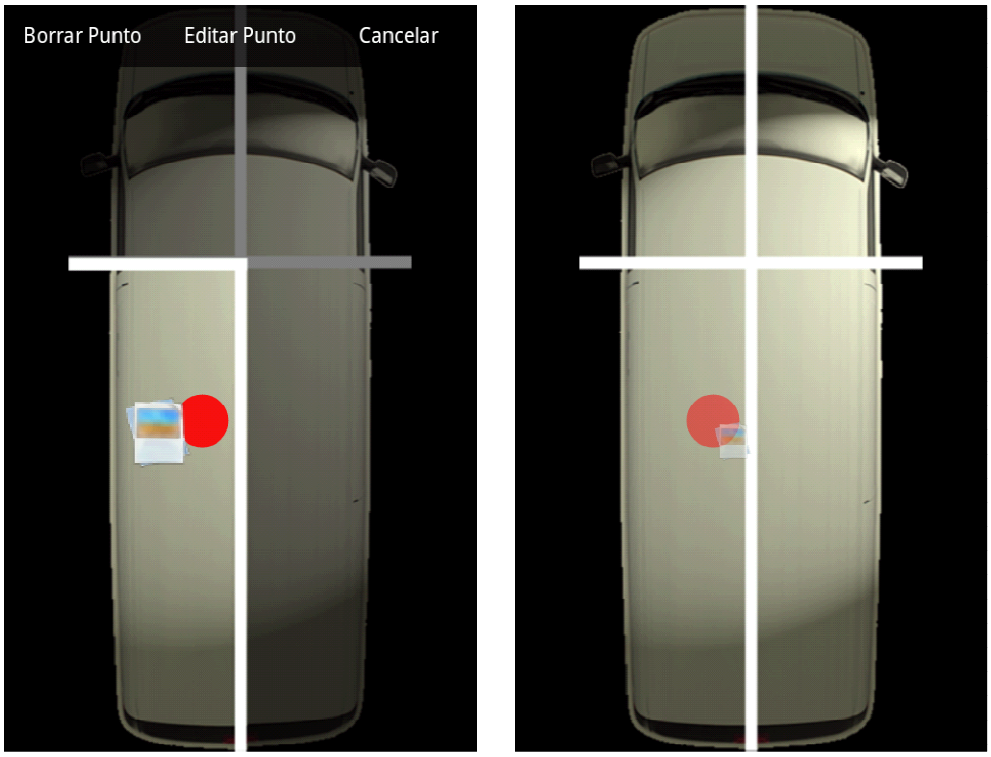


Fig. 17 Punto seleccionado y sin seleccionar

Cuando el punto está seleccionado (como en la imagen de la izquierda de la imagen anterior) al tocar la vista de la fotografía saltará el evento *onTouchEvent*() y nos mostrará la fotografía.

FormInicio y FormCrearProyecto

Comenzamos a analizar las clases *Activity* creadas. Conviene recordar que estas clases equivalen a una pantalla de las que ve el usuario. Las dos clases siguientes (figura 18) son clases propias que heredan de la clase *Activity*, la primera de ellas *FormInicio* es la *Main Activity* es decir la que se lanza cuando se abre la aplicación, y la segunda es la pantalla de creación de un nuevo proyecto.

Dentro de la primera clase están definidas dos clases auxiliares (*AdaptadorProyectos* y *AdaptadorTitulares*) que son adaptadores que heredan de la clase *ArrayAdapter*, en Android para crear *ListViews* personalizadas debemos trabajar con adaptadores. Éstos proporcionarán a la lista los datos a mostrar, y podemos personalizarlos creando adaptadores con datos más elaborados que un simple texto. **[6]**

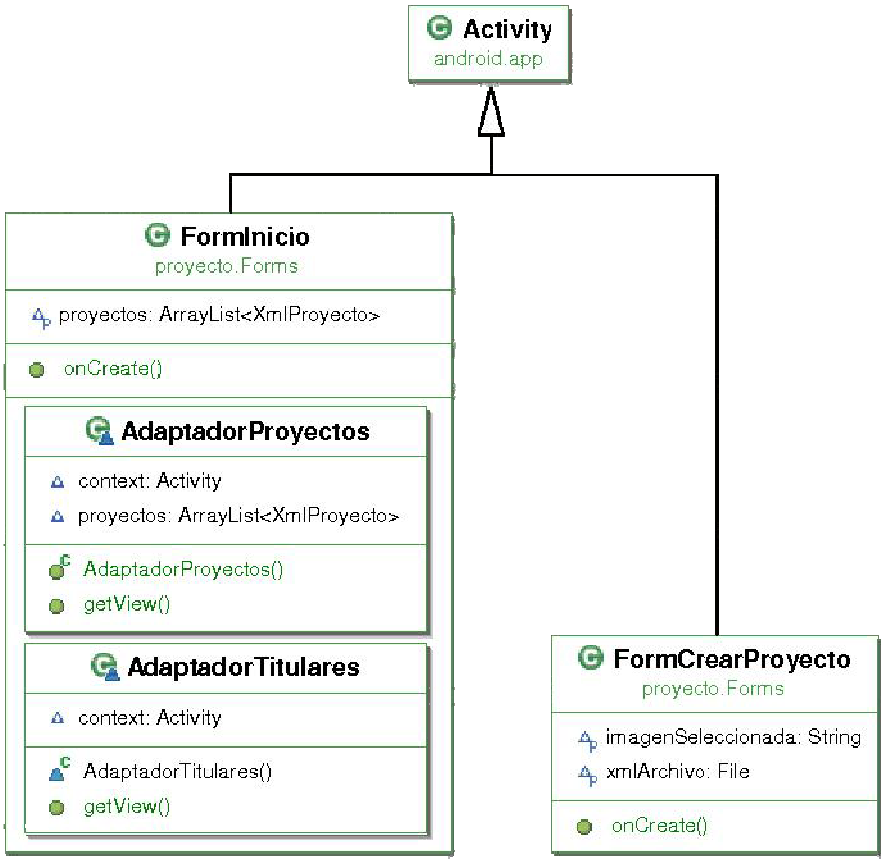


Fig. 18 Clases FormInicio y FormCrearProyecto

En nuestro caso cada adaptador mostrará una lista diferente de datos. Mientras que el *AdaptadorTitulares* dispone de las dos opciones principales Nuevo Proyecto y Abrir Proyecto, el *AdaptadorProyectos* se encargará de rellenar la lista de proyectos existentes.

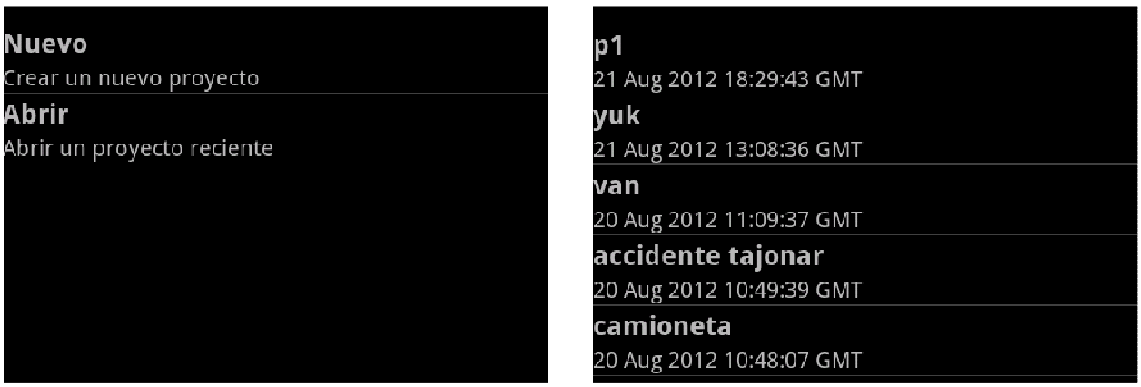


Fig. 19 Adaptaders

Cada vez que se cree un objeto de estas clases el evento será capturado por *onCreate*(), donde cada clase obtendrá el layout correspondiente de los recursos, donde previamente se definió en formato XML. A modo de ejemplo podemos ver el código XML que define el Layout que usa como vista la clase *FormCrearProyecto*:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
      android:id="@+id/widget32"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:orientation="vertical"
      xmlns:android="http://schemas.android.com/apk/res/android">
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"

    android:textSize="16dp"
    android:layout_marginTop="10dp"
    android:layout_marginBottom="10dp"
    android:text= "@string/Nuevo"/>

<EditText
    android:id="@+id/editTextNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:singleLine="true">
</EditText>

    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:textSize="16dp"
    android:layout_marginBottom="10dp"
    android:text= "Seleccione plantilla de vehículo:"/>

    <Gallery
        android:id="@+id/gallery1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="10dp"/>

    <ImageView
        android:id="@+id/imagenCoche"
        android:layout_width="wrap_content"
        android:layout_height= "200dp"
        android:src="@drawable/sample_1"
         android:layout_marginBottom="10dp" />

    <Button
        android:id="@+id/buttonGuardar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Guardar" />

</TableLayout>
```

Podemos identificar elementos como dos *TextView* (vista estándar de texto), *Gallery* (donde estará la galería de siluetas), una *ImageView* (donde se mostrará las imágenes de la silueta en grande conforme el usuario vaya seleccionándolas en la galería) y un botón de guardar proyecto que nos llevaría a la siguiente actividad: *FormProyecto*.

FormProyecto:

La siguiente clase es quizás la más importante de todas las vistas hasta ahora, puesto que se encargará de gestionar la funcionalidad básica de la aplicación.

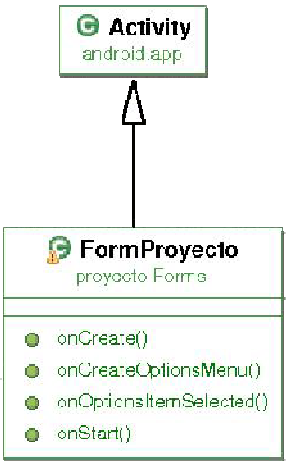


Fig. 20 FormProyecto

Esta *Activity* se mostrará tras crear o abrir un proyecto, recibirá pues el proyecto como un objeto de la clase *XMLProyecto*, leerá y dibujará sus puntos, controlará la creación de puntos, los cambios de gravedad, llamara a la cámara fotográfica etc.. Esto lo podemos ver mejor si mostramos los métodos privados de la actividad, los que usa internamente para dar toda la funcionalidad como en la figura 21.

Fig. 21 FormProyecto con sus métodos privados

Entre los métodos podemos observar que existen *lanzarGPS(..)* y *ObtenerPosicion(..)*, éstos sirven para obtener la posición GPS. Esta funcionalidad se añadió al final, no está definida en los requisitos, no obstante parece interesante almacenar la posición GPS del accidente. Para ello *FormProyecto* hace uso de una clase que hereda de la clase *LocationListener*, y es la siguiente:

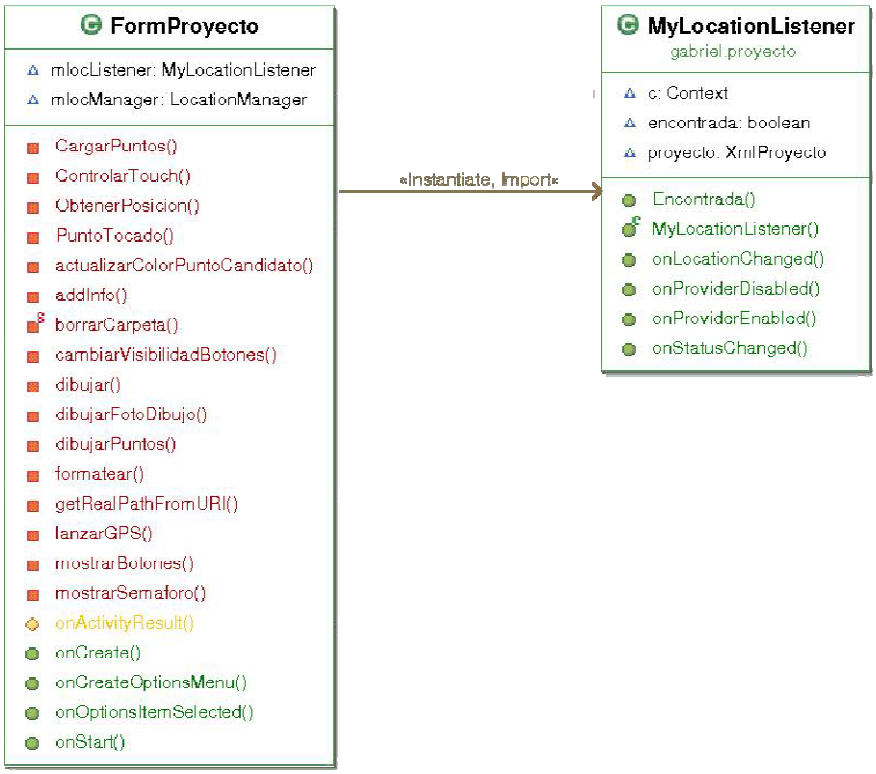


Fig. 22 MyLocationListener

Es una clase que hará de *Listener* para el GPS, y sólo será llamada con este fin. Una vez encuentre la posición la escribirá en el proyecto.

FormPunto:

Ésta clase no fue definida en el análisis, porque surgió de un detalle no tenido en cuenta a la hora de pensar la aplicación. En el momento de desarrollo opté por la opción de crear esta clase porque parecía la opción más intuitiva de cara al usuario. Veremos su uso mejor detallado en el manual de usuario (tema 8), no obstante trataré de explicar en las siguientes líneas, el porqué de la decisión de añadir esta *Activity*.

En cuanto creamos un punto pulsando sobre el layout de la actividad *FormProyecto*, se nos muestra el círculo y un semáforo para que elijamos la gravedad; además aparecerán tres botones. Uno de los botones sirve para cancelar (en cuyo caso el punto creado no se guardaría), para añadir una fotografía al punto, y para aceptar; éste último botón guardará en ese momento el punto, con el color presionado del semáforo, y con la foto si el usuario la hubiese añadido.

Por otro lado el usuario puede tocar la pantalla para seleccionar un punto, podríamos optar por mostrar los mismos botones que en la creación de un punto incluyendo el semáforo, sin embargo esto no sería una buena opción. En primer lugar puede que el usuario quiera borrar ése punto, por lo que necesitaríamos un botón de borrado, también puede que sólo desee consultarlo (ver su fotografía), en ninguno de los dos casos sería adecuado mostrar el semáforo para reeditar la gravedad, tampoco el botón de añadir fotos. O por otra parte puede realmente querer editar dicho punto, entonces si sería correcto mostrar el semáforo y el botón.

Por eso la opción final fue la siguiente: cuando el usuario selecciona un punto, el punto se dibuja como seleccionado, pudiendo acceder a su fotografía. Aparecerán tres botones, se mantiene el botón cancelar, que no hará otra cosa que deseleccionar el punto; tendremos un botón para borrar el punto, y por último un botón para editar el punto. El botón editar será el que lance la actividad *FormPunto*, desde la cual podemos cambiar la gravedad y la imagen, y podemos guardar o desechar los cambios efectuados.

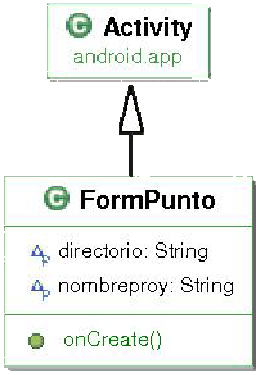


Fig. 23 FormPunto

6.2.2 Relaciones entre clases

A continuación vamos a explicar las relaciones e interacciones entre las distintas clases.

Para comenzar, a pesar de que no se trate de un tipo de asociación se explicará brevemente como pasamos de una actividad, y como se comunican entre ellas. Esto se ha implementado con *Intents* (esta clase ya se explicó en el tema 2.2 Conceptos básicos de Android), que nos permiten llamar de una actividad a otra, pasarle información y esperar una respuesta.

En el siguiente ejemplo vemos cómo desde la clase *FormInicio*, cuando abrimos un proyecto existente se crea un *Intent* para lanzar la actividad *FormProyecto*.

```
XmlProyecto xmlP=proyectos.get(position);
Intent intent=new Intent(FormInicio.this,FormProyecto.class);
intent.putExtra("Nombre", xmlP.getNombre());
```

```
intent.putExtra("Silueta",xmlP.getSilueta());
intent.putExtra("Directorio",
Directorio.toString()+"/"+xmlP.getNombre());
startActivityForResult(intent,100);
```

Con la función *putExtra()* le pasamos a información a la nueva Actividad y esperamos un resultado. La otra actividad *FormProyecto* recogerá los datos de la siguiente manera:

```
Bundle bundle = getIntent().getExtras();
nombreproy = bundle.getString("Nombre");
String silueta = bundle.getString("Silueta");
Directorio = bundle.getString("Directorio");
```

Luego con el método *onActivityResult(…)* se recoge el evento de finalización de la otra actividad, que antes de finalizar con *finish()* podrá escribir un resultado para que la actividad anterior lo maneje.

Una vez vista la comunicación entre las clases *Activity* vamos a ver las relaciones de asociación que existen entre las clases que se han creado para la aplicación.

FormProyecto:

Como ya habíamos explicado en el capítulo anterior en la definición de las clases, la clase *FormProyecto* tiene un peso muy relevante en nuestra aplicación. Esto se comprende mejor cuando observamos el diagrama de la figura 24.

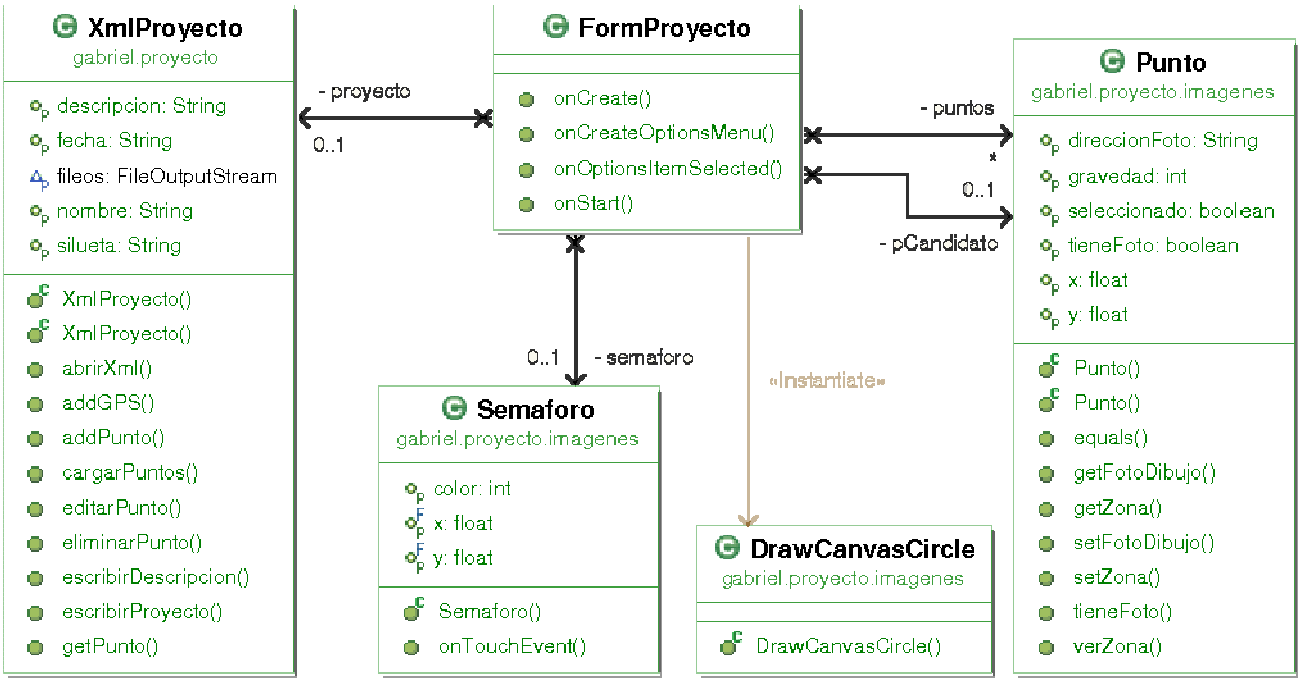


Fig. 24 Asociaciones de FormProyecto

Como podemos observar cada instancia que creemos de *FormProyecto* va a asociado a un objeto de la clase *XmlProyecto*, es decir a un proyecto concreto. Habrá recibido a través de un *Intent* el nombre del proyecto que hay que abrir. Con el nombre inicializará el atributo *proyecto*, y de ahí leerá todos los datos que necesita, como la silueta para mostrar de fondo (dividida en zonas), los puntos que posee el proyecto (si es que los posee), cambiará la descripción del proyecto cuando se requiera etc.

Se observa una asociación de *Semaforo* con *FormProyecto* a través de atributo. La cardinalidad de la asociación es 0..1, esto es debido a que sólo mostrará el semáforo cada vez que se crea un punto, y hasta que no se haya guardado o descartado éste, no se puede crear ni seleccionar otro punto; por lo que sólo habrá un semáforo en pantalla como máximo. Basta con crear un objeto y usarlo cada vez que se cree un punto. El color elegido se guardará en cada punto.

Existen dos asociaciones con la clase Punto, una mediante el atributo *puntos* que vemos que tiene de cardinalidad entre 0 y n puntos, y un atributo llamado *pCandidato*, que solo puede ser 0 o 1. En el atributo *puntos* se guardan todos los puntos del proyecto, éstos se recogen usando la función *cargarPuntos()* de la clase *XmlProyecto*. La otra asociación entre ambas clases es un punto *pCandidato* en donde guardamos el punto que aún no se ha guardado en memoria, puesto que acaba de ser creado y se está editando. Si el usuario acepta guardar el punto, se añadirá el punto *pCandidato* a *puntos* y al archivo XML del proyecto, *pCandidato* entoncés pasará a valer *null.* La clase punto posee un atributo boléano *Seleccionado* que indicará a la actividad si el punto está seleccionado o no.

Por último la clase *FormProyecto* es la que dibuja los puntos a través de objetos *DrawCanvasCircle*, cada punto guardará luego su vista correspondiente para que pueda ser borrado de la pantalla cuando se requiera. Para entender mejor esto veremos algunas de las asociaciones de la clase *Punto*.

Punto:

Como hemos comentado antes, *FormProyecto* dibuja el punto y guarda la vista en la clase punto. Es el atributo *vista* de tipo *View* (figura 25).

Por ejemplo, cada vez que el usuario seleccione un punto, la actividad deberá recorrer toda su colección de puntos, comprobar cual es igual con la función sobrescrita de *Punto.equals()*, borrar la vista de dicho punto (que ya estaría dibujada) y dibujarla más intensa, para que el usuario note que el punto está seleccionado. De igual modo al deseleccionar el punto, la vista se hará de un color menos intenso.

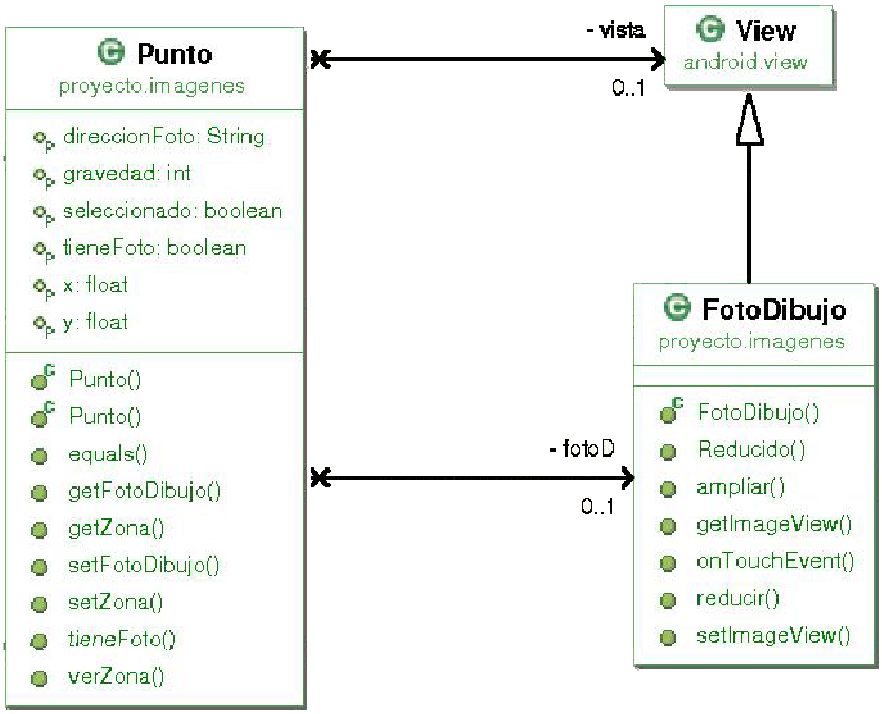


Fig. 25 Asociaciones de la clase punto

Además un objeto de la clase *Punto* tiene como atributo una imagen, con su dirección, y además un objeto de la clase *FotoDibujo*. Al tratarse de un objeto que hereda de *View*, *FormProyecto* dibujará el icono de la fotografía al lado del punto accediendo a través del propio punto. El evento *onTouchEvent()* hará que se muestre la foto cuando se presione en el icono, siempre que éste este agrandado (es decir que se acabe de crear, o seleccionar un punto).

6.2.3 Diagrama de clases completo

Tras haber analizado con detenimiento las clases más relevantes, con sus atributos y métodos más importantes, así como el comportamiento entre ellas, sólo queda mostrar el diagrama de clases completo en la figura 26.

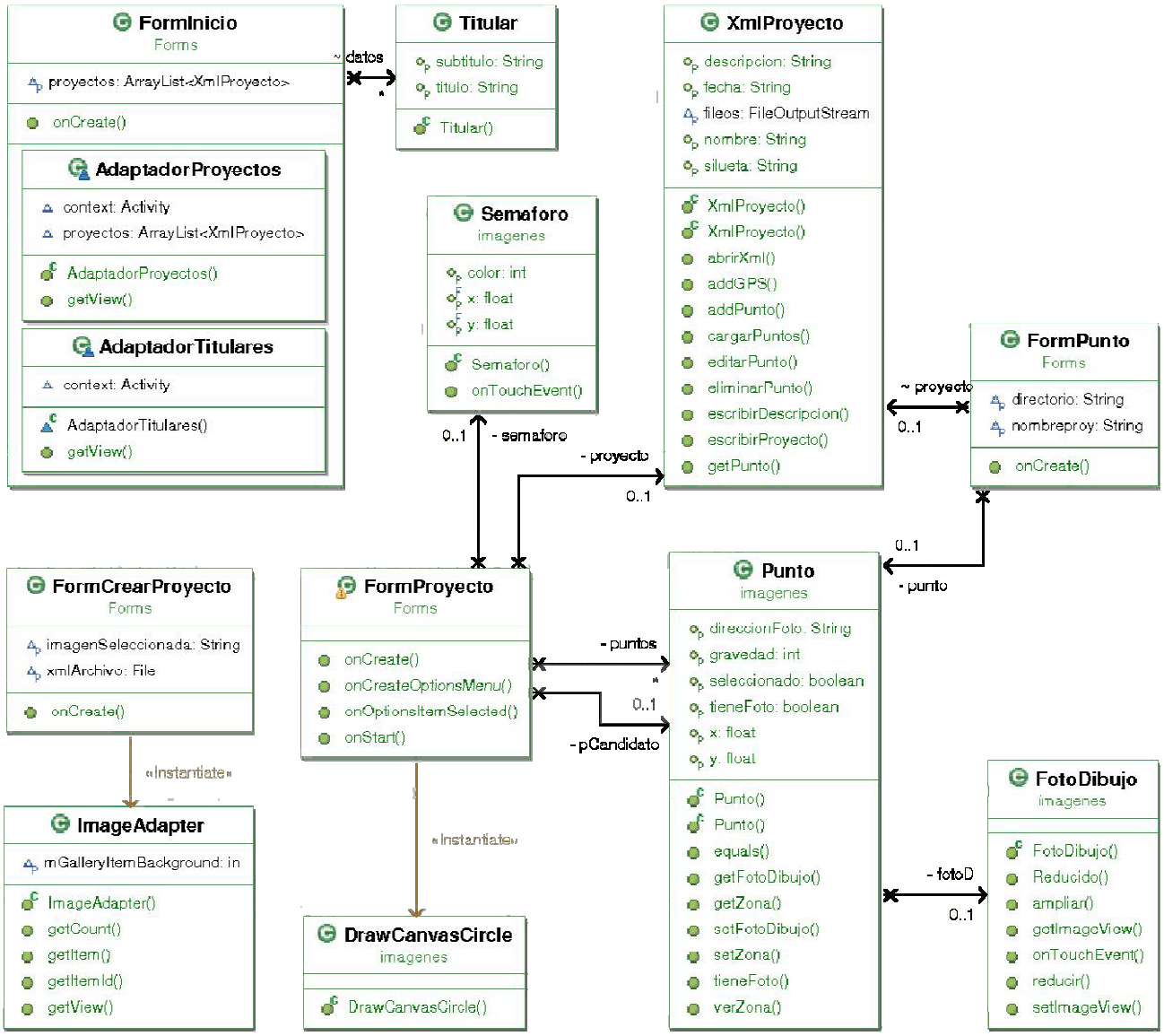


Fig. 26 Diagrama de clases completo

Se dividieron las clases en tres paquetes, uno para las actividades denominado *Forms*, otro para todo lo que tiene que ver con el tratamiento de imágenes y otro para el resto, es decir, las clases *XmlProyecto*, y los adaptadores. El diagrama de paquetes lo podemos ver en la figura 27.

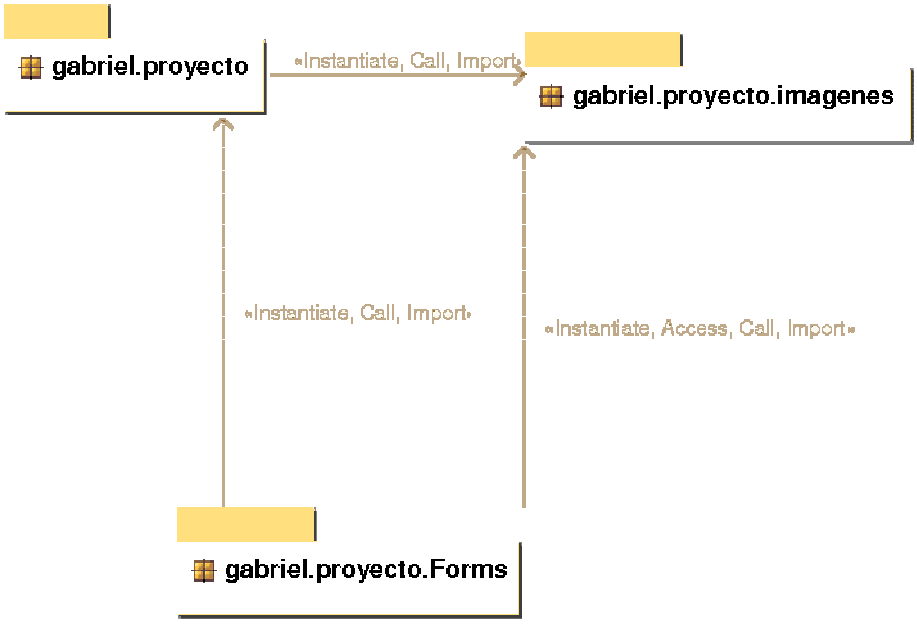


Fig. 27 Dependencias entre paquetes

Las clases se dividen en tres paquetes, en el primero de ellos tenemos a la clase *XmlProyecto* junto a alguna clase auxiliar. En el nombrado *Forms* se encuentran todas las clases que heredan de la clase *Activity*, es decir, las pantallas de la aplicación. Y por otro lado, en imágenes están las clases que muestran o poseen imágenes o vistas. Es decir, la clase *Semaforo, Punto, FotoDibujo* etc. En el diagrama anterior podemos ver como interactúan los paquetes entre ellos. Las actividades son las que instancian las demás clases cuando las requieren.

6.3 Detalles técnicos

En este subcapítulo se explicarán dos detalles técnicos de la implementación. El primero de ellos será el trabajo con *Views*, y el otro la lectura y escritura de archivos locales XML en Android. Se han elegido estos dos porque son los que más funcionalidad dan al conjunto de la aplicación. El primero, el trabajo con vistas, definirá el grado de usabilidad de la aplicación, mientras que el segundo es necesario para el almacenamiento de los datos recogidos. Nos se mostrará todo el código implementado, tan sólo algunas partes que he considerado interesantes.

6.3.1 Trabajo con Views

La clase *View* es el bloque básico de una interfaz de usuario *UI*, se trata de un área rectangular que es dibujada y maneja eventos. Como vemos la descripción es muy amplia y es que todos los *widgets* de Android (botones, textos etc.) heredan de ésta clase. También es posible crear nuestras vistas personalizadas. Una de las maneras de hacer esto es sobrescribir en la clase personalizada (que herede de *View*) el método *onDraw(Canvas c)*, de modo

que podemos controlar el momento antes de ser dibujada la vista y cambiarla. La clase que recibimos *Canvas* se podría decir que es el lienzo, a éste le podemos asociar un objeto *bitmap* que dibujar. La función *invalidate()* de una *View* sirve para forzar a que ésta vuelva a dibujarse.

En la aplicación desarrollada ha existido un trabajo importante con objetos que heredan de la clase *View*. La actividad principal consiste en una vista de fondo, que además cambia según la zona presionada (recordemos la figura 9), sobre la cual se irán añadiendo diferentes vistas, básicamente puntos dibujados con su correspondiente color o gravedad, y sus íconos de imágenes añadidas. A modo de ejemplo, para no saturar al lector, se explicara como se ha implementado la vista del semáforo y como se dibujan los puntos.

Como se ha explicado previamente, el semáforo se muestra apagado al crear un punto, el usuario podrá cambiar su color dinámicamente presionando en cualquiera de las tres luces, se encenderá la luz correspondiente y el punto cambiará de color, pudiendo también apagarlo de nuevo al volver a presionar sobre la luz encendida.

El siguiente código es el de construcción del semáforo. En nuestros recursos está la imagen del semáforo (figura 6) previamente añadida.

```
public class Semaforo extends View{

      private float x;
      private float y;
      private  Bitmap bmap;
      private int color;


   public Semaforo(Context context, float x, float y) {
       super(context);

       this.x = x;
       this.y = y;

       bmap = BitmapFactory.decodeResource(getResources(),
      R.drawable.semaforo);
       this.color=0;


   }
   @Override
   protected void onDraw(Canvas canvas) {
         super.onDraw(canvas);
         canvas.drawBitmap(bmap, this.x , this.y, null);
        }
```

Muy sencillo hasta el momento, la llamada al semáforo deberá ser con las coordenadas. Habrá que elegir unas coordenadas de modo que el semáforo se muestre cercano al punto pero sin taparlo, además que no debe exceder de la pantalla. Vamos a ver, a continuación, como se muestra el semáforo para un punto. Esto se hace desde la actividad *FormProyecto*, y conviene fijarse en

que la variable *semaforo* se sobrescribe de modo que sólo habrá una vista del semáforo en pantalla como máximo. En función de la coordenada *Y* mostramos el semáforo más alto o más bajo.

```
private void mostrarSemaforo(Punto punto) {
      // Muestra el semaforo cuando se toca un punto.
            if (punto.getY() < 0.5F) {
            semaforo = new Semaforo(context, punto.getX() * ancho+ 5,
                        punto.getY() * altura);
      } else {
            semaforo = new Semaforo(context, punto.getX() * ancho + 5,
                        (punto.getY() * altura) - (0.2F * altura));
      }
      FrameLayout main =(FrameLayout) findViewById
(R.id.formproyecto_view);

      Toast toast = Toast.makeText(context,
            "Gravedad del punto seleccionado",Toast.LENGTH_SHORT);
      toast.show();
       main.addView(semaforo);
      }

private void actualizarColorPuntoCandidato() {

            FrameLayout main = (FrameLayout)
findViewById(R.id.formproyecto_view);
            main.removeView(pCandidato.getVista());
            pCandidato.setGravedad(semaforo.getColor());
            dibujar(pCandidato, INTENSO);
}
```

Además está la función *actualizarColorPuntoCandidato()*, a la que llamaremos al tocar la pantalla, cuando un punto acabé de crearse y aún no se haya guardado. En ese momento puede que el usuario haya cambiado el color del semáforo, y habrá que actualizar el color del punto, así la aplicación luce más dinámica. El cambio de color del semáforo lo gestionará la propia clase semáforo de la siguiente forma:

```
@Override
public boolean onTouchEvent(MotionEvent event)
{

Rect rectangulo=new Rect();
rectangulo.set((int)this.x, (int)this.y, (int)(this.x +
bmap.getWidth()),(int)(this.y + bmap.getHeight()));


if (rectangulo.contains((int)event.getX(), (int)event.getY())){

      Rect rectRojo=new Rect(rectangulo.left, rectangulo.top,
      rectangulo.right,rectangulo.bottom-2*(rectangulo.height()/3));

      Rect rectAmbarRojo=new Rect(rectangulo.left, rectangulo.top,
      rectangulo.right,rectangulo.bottom-1*(rectangulo.height()/3));

      if (rectRojo.contains((int)event.getX(), (int)event.getY())){
```

```
            if (color==1){

                  bmap=BitmapFactory.decodeResource(getResources(),
                  R.drawable.semaforo);

                  this.color=0;

                     }else{

                  bmap=BitmapFactory.decodeResource(getResources(),
                  R.drawable.semaforo_rojo);

                  this.color=1;
                  }
      }
      else if(rectAmbarRojo.contains((int)event.getX(),
      (int)event.getY())){

            if (color==2){

                  bmap=BitmapFactory.decodeResource(getResources(),
                  R.drawable.semaforo);

                  this.color=0;

                  }else{

                  bmap=BitmapFactory.decodeResource(getResources(),
                  R.drawable.semaforo_ambar);

                  this.color=2;
                  }
              }else{

                  if (color==3){

                  bmap=BitmapFactory.decodeResource(getResources(),
                  R.drawable.semaforo);

                  this.color=0;

                  }else{

                        bmap=BitmapFactory.decodeResource(
                        getResources(),R.drawable.semaforo_verde);

                        this.color=3;
                  }
               }

      invalidate();
      }

return false;

}
```

Para controlar que parte del semáforo se está tocando se ha usado la clase *Rect* que equivale un rectángulo y posee un método para ver si un punto definido con dos coordenadas X e Y es contenido por el rectángulo.

El primer rectángulo creado corresponde al propio semáforo, así comprobamos que el usuario haya tocado dentro de éste. A continuación creamos un rectángulo que equivale al rectángulo de la luz roja, y otro que equivale a la luz roja y ámbar unidos. Con esto valdrá para saber en que luz se está tocando. Si el rectángulo rojo contiene las coordenadas, significa que se ha tocado la luz roja; en ese caso veremos el color que posee el semáforo, si no es el rojo, se cambiará el *bitmap* al *bitmap* con el semáforo rojo encendido, en caso contrario, el *bitmap* será el del semáforo apagado. Actualizaremos también el atributo color. Haremos lo mismo con el otro rectángulo, el ámbar y rojo, si contiene las coordenadas cambiaremos al color necesario, y sino las contiene el usuario habrá presionado al color verde y se tratará de igual manera.

Tras las comprobaciones se llama a *invalidate()* que como se ha explicado forzará a redibujar la vista. Como la dirección del *bitmap* se ha cambiado, al volver a dibujar el semáforo hará el efecto de encenderse la luz presionada (o apagarse si hemos presionado sobre una luz encendida). Por último devolveremos *false*. En función de lo que devuelva la función *onTouchEvent(..)* se considerará el evento manejado o no, si devuelve true el evento se considerará manejado, mientras que si devuelve *false*, pasará a la vista padre. En este caso, la vista padre es la *contentView* que deberá cambiar el color del punto cuando se cambie el del semáforo, por tanto, es necesario que devuelva *false*, para que el evento vuelva a la vista padre.

Por último parece interesante mostrar como se dibujan los puntos, puesto que es una de las funciones más usadas en la aplicación. Esta función se encuentra en la clase *FromProyecto* y se usará tanto para crear un nuevo punto, como al seleccionar uno existente.

```
private void dibujar(Punto punto, int Alpha) {


View v = new DrawCanvasCircle(context, punto.getX() * ancho,
  punto.getY() * altura, altura / 28, Alpha, punto.getGravedad());
punto.setVista(v);

FrameLayout main= (FrameLayout) findViewById(R.id.formproyecto_view);

main.addView(v);

// Si el punto posee una foto
if (punto.tieneFoto()) {
      FotoDibujo fd = punto.getFotoDibujo();

      if (fd == null) {
            dibujarFotoDibujo(punto, punto.getDireccionFoto());
            fd = punto.getFotoDibujo();
      }

      // Si ésta esta reducida ya, no la reduzco más
      if (fd.Reducido()) {
            // A no ser que sea el punto recien creado o seleccionado
            if (Alpha == INTENSO) {
                  punto.getFotoDibujo().ampliar();
            }
```

```
            } else {
            if (Alpha != INTENSO) {
                  punto.getFotoDibujo().reducir();
            }
      }
}
```

El código puede parecer algo confuso sin embargo es muy sencillo. Dibujar toma dos parámetros, el punto y un entero *Alpha*, éste entero representa la intensidad de color del punto. Esto es debido a que cuando un punto ha sido creado o seleccionado lo dibujaremos más intenso, mientras que si lo dibujamos de fondo estará más claro. El atributo *main* es la vista principal a ella le añadimos la vista creada con la clase *DrawCanvasCircle()* . Sólo queda ver si el punto posee imagen para dibujar la vista del ícono. En el caso de que posea imagen cogeremos su ícono: un atributo del objeto *Punto* de tipo *FotoDibujo*. Si es igual a null (por ejemplo cuando cargamos directamente los puntos del Xml aún no se ha creado la clase FotoDibujo), debemos dibujar el icono con la función *dibujarFotoDibujo(Punto p,String direccion)*. La función propia *reducir()* de la clase *FotoDibujo* reducirá el ícono de la imagen (ver fig. 17), si está reducida y se ha ordenado dibujar el punto con intensidad alta, es decir el punto se ha creado o seleccionado, el ícono deberá ser grande, mientras que si el punto se acaba de guardar o de cargar el ícono será pequeño. La clase FotoDibujo mostrará la foto al pulsar sobre el icono únicamente si este se encuentra ampliado.

6.3.2 Tratamiento de datos XML

Existen varias formar de escribir y leer archivos XML de nuestro directorio en Android **[7]**. Para esta aplicación se han usado dos técnicas, una de ellas para crear archivos XML rápidamente es mediante el serializador *XmlSerializer*, y la otra, que usaremos para leer y editar los diferentes archivos XML es usando DOM. Toda la funcionalidad de lectura y escritura de los datos está implementada en la clase *XmlProyecto* (ver tema 6.2.1).

Diferenciamos pues, entre crear un nuevo proyecto y leer uno existente. También en el código vemos estas diferencias, la clase dispone de dos constructores diferentes.

```
public class XmlProyecto {

      private String nombre;
      private String silueta;
      private String fecha;
      private String descripcion;
      private String direccion;
      XmlSerializer serializer;
      FileOutputStream fileos;
      Document documento;

      static private String[] Siluetas = { "Coche normal", "Van",
      "Camioneta","Minibus", "Camion" };
      static private String[] Zonas = { "Piloto", "Copiloto",
      "Asiento izquierdo", "Asiento derecho", "Central"};
```

```
      public XmlProyecto(File archivo) throws Exception {

            this.direccion = archivo.getCanonicalPath();
            fileos = new FileOutputStream(archivo);

// creamos un XMLSerializer para escribir el proyecto mas adelante

            serializer = Xml.newSerializer();
            this.silueta = null;
      }

      public XmlProyecto() {

      }
```

El primer constructor lanza una excepción, en el caso de que el fichero no se pueda abrir, y deja preparado el objeto serializer de tipo *XmlSerializer*. El segundo constructor está vacío y está pensado para abrir un archivo XML existente. Usaremos el *XmlSerializer* únicamente para escribir un nuevo proyecto, debido a que es muy sencillo:

```
public void escribirProyecto(String nombre, String silueta)
                  throws Exception {

            serializer.setOutput(fileos, "UTF-8");
            serializer.startDocument(null, Boolean.valueOf(true));
            serializer.setFeature(
            "http://xmlpull.org/v1/doc/features.html#indent-output",
            true);

            serializer.startTag(null, "Proyecto");

            serializer.startTag(null, "Nombre");
            serializer.text(nombre);
            serializer.endTag(null, "Nombre");
            serializer.startTag(null, "Fecha");
            Date date = Calendar.getInstance().getTime();
            serializer.text(date.toGMTString());
            serializer.endTag(null, "Fecha");

            ..............................

            serializer.endTag(null, "Proyecto");
            serializer.endDocument();

            // escribe los datos en el FileOutputStream
            serializer.flush();
            fileos.close();

            this.nombre = nombre;
            this.silueta = silueta;
            this.fecha = date.toGMTString();
            this.descripcion = null;
      }
```

Las etiquetas se crean en el orden con el que se va llamando a las funciones *startTag* y *endTag*, que abren y cierran las etiquetas añadiendo texto en medio con la función *Text*. Para finalizar se escriben los datos en el

FileOutputStream y se cierra. No obstante éste método no es versátil, simplemente nos vale para escribir un proyecto nuevo. En el momento que queremos añadir una etiqueta *Punto* a la etiqueta *Puntos*, o añadir una descripción al proyecto se necesita otro método de escritura.

Para esta funcionalidad se ha implementado DOM (*Document Object Model*). El modelo DOM ofrece una serie de clases y métodos para la lectura y escritura. Lo primero que hace es leer todo el fichero Xml y elabora un árbol sobre el que podremos navegar. En nuestro caso el árbol creado tendrá como nodo raíz el proyecto y como hijos cada uno de los atributos del proyecto (nombre, silueta, un subárbol de puntos etc.). A continuación vemos como se abre un proyecto existente, es decir un archivo Xml.

```
public boolean abrirXml(String direccion) {
try {
      this.direccion = direccion;
      DocumentBuilderFactory factory = DocumentBuilderFactory
                              .newInstance();

      DocumentBuilder builder = factory.newDocumentBuilder();
      FileInputStream filename = new FileInputStream(direccion);

      this.documento = builder.parse(filename);
      this.documento.normalize();
      NodeList nodos = documento.getElementsByTagName("Nombre");
      this.nombre = nodos.item(0).getTextContent();
      nodos = documento.getElementsByTagName("Fecha");
      this.fecha = nodos.item(0).getTextContent();

      nodos = documento.getElementsByTagName("nombreSilueta");

      this.silueta = nodos.item(0).getTextContent();

      try {
            this.descripcion = documento.getElementsByTagName(
                  "DescripcionProy").item(0).getTextContent();

      } catch (Exception e) {
            this.descripcion = "";
      }
      return true;
      }
catch (Exception ex) {

      return false;

}
```

Para añadir o editar información también se ha usado DOM, a modo de ejemplo vamos a ver como se añade la descripción a un proyecto XML. Para guardar los cambios en el documento hacerlo debemos usar la clase *Transformer*.

```
public boolean escribirDescripcion(String des) {
try {
      Node nodoProyecto = documento.getElementsByTagName("Proyecto")
            .item(0);
```

```
        try {
       // Si existe la descripcion la cambio
             Node Descripcion = documento.getElementsByTagName(
                   "DescripcionProy").item(0);
             Descripcion.setTextContent(des);
             }
       catch (Exception ex) {
       // Si no existe creo el nodo y lo añado
             Element desc = documento.createElement("DescripcionProy");

             desc.appendChild(documento.createTextNode(des));
             nodoProyecto.appendChild(desc);

             }

Transformer transformer = TransformerFactory.newInstance()
                              .newTransformer();

transformer.setOutputProperty(OutputKeys.INDENT, "yes");

StreamResult result = new StreamResult(new File(direccion));
DOMSource dom = new DOMSource(documento);
transformer.transform(dom, result);
this.descripcion = des;
return true;


} catch (Exception ex) {
      return false;
}
}
```

Si el nodo ya existe basta con cambiar en contenido de éste con el método *setTextContent()*, en caso contrario creamos una variable de tipo *Element* en el documento abierto y hacemos un *append* al hijo deseado en éste caso el nodo raíz. De igual modo, cuando añadamos un punto lo haremos al nodo *Puntos*.

7. Pruebas

Para realizar las pruebas se han utilizado diferentes dispositivos. La mayoría de las pruebas se han realizado desde dispositivos virtuales de Android. La SDK de Android nos permite crear dispositivos virtuales, asignarles el hardware y la versión de Android, e instalar y testear nuestras aplicaciones en ellos. Además con el fin de probar la aplicación se me proporcionó un dispositivo móvil, un modelo chino con la versión 2.2 de Android. También se probó el móvil para un dispositivo Samsung Galaxy S GTI-9003, con Android 2.3.6 y para otro modelo htc desire, android 2.3.3. Era necesario probar en diferentes móviles de diferentes resoluciones para ver si la proporción de los puntos y las siluetas se mantenía (es decir, el programa identificaba correctamente la zona del vehículo pulsada, detectaba si estaba o no dentro de los límites de la silueta etc.).

En vista de que el desarrollador, ósea yo, carecía de experiencia en el desarrollo de aplicaciones Android y con el fin de simplificar y cometer menos errores, las funcionalidades importantes se programaron en programas diferentes que después se probaron e integraron. Los diferentes programas eran: un programa de manejo de listas o *ListView*, otro para la obtención de fotos (cámara y galería), otro de imágenes (dibujar puntos y semáforos) y por último otro programa con el esqueleto de la aplicación sin desarrollar. Tras probar cada programa por separado se unieron todos ellos en la aplicación definitiva añadiéndole por último la funcionalidad de almacenar y leer los datos.

A continuación se mostrarán una serie de pruebas que se desarrollaron tras la integración de los programas citados anteriormente y la implementación. No obstante, el código se fue probando y revisando conforme se iba desarrollando, por lo que sería imposible mostrar todas las pruebas realizadas.

Prueba de escritura

El nombre del proyecto debe ser único y sirve de identificador. Como ya se ha explicado el nombre se usará para crear una carpeta en la *SDCard* de igual nombre, por tanto era importante realizar pruebas en este punto crítico. Para ello se probó a crear proyectos con caracteres extraños, palabras muy largas, saltos de línea y espacios. Los espacios entre palabras son sustituidos por caracteres ‘+’, no se permiten saltos de línea y los caracteres extraños son admitidos así como las palabras muy largas.

Prueba de carga

Otra parte importante a probar es el hecho de cargar una cantidad importante de puntos. En estos puntos añadiremos imágenes que se almacenarán en memoria. Por tanto habrá que ver que sucede cuando la memoria se llene. Para ello creamos un dispositivo virtual con el mínimo de capacidad disponible, en este caso 9MB. Lo llenamos casi al completo y entonces cuando vamos a añadir una imagen al proyecto vemos que el propio sistema al llamar al *Intent* de la cámara ya nos devuelve el mensaje que la

tarjeta está llena. Además el número de proyectos es suficientemente grande ya que ocupan en memoria muy poco, siendo las imágenes lo que más ocupa.

Otra prueba realizada respecto a la carga es la de añadir un gran número de puntos a un proyecto. Al abrir el proyecto de nuevo hay que comprobar que el dispositivo no se bloquee, que todos los puntos se dibujen correctamente justo al crear la *Activity*, y que no haya colisiones mientras se carga. Esto se comprobó que funcionaba bien, y una de las razones por la que así fue es por el modo de dibujar los puntos al abrir el proyecto:

Para que haya independencia entre la creación de la actividad y la carga de puntos se ha usado la clase *Handler*. Dentro de un hilo lo que hacemos es decir al sistema con el método *post* que meta en la cola de procesos el dibujar los puntos.

```
@Override
      public void onStart() {
            super.onStart();
            handler = new Handler();
            new Thread() {
                  @Override
                  public void run() {

                        handler.post(new Runnable() {
                              @Override
                              public void run() {

                                    dibujarPuntos();
                              }
                        });
                  }
            }.start();
}
```

Se podrían hacer muchas más pruebas de carga, y de otros tipos, para ver como se comportaría la aplicación ante situaciones límite (fin de batería, llamadas entrantes...). Sin embargo sabemos que los datos se almacenan constantemente cada vez que se crea un punto, por lo que no tenemos peligro de perdida de datos por lo que parece suficiente con esto.

8. Manual de usuario

Ya se ha explicado en que consiste la aplicación y que pasos se han seguido para desarrollarla. A continuación se va a mostrar como usar la aplicación, de modo que el usuario final posea toda la información necesaria para recoger datos de los diferentes siniestros. Por tanto este tema será puramente práctico y se acompañará la información con imágenes para hacerlo más comprensible.

Creación de un proyecto:

Lo primero que haremos si queremos recoger datos de un siniestro es crear un nuevo proyecto, lo cual nos generará una escena sobre la que trabajar, y donde los datos que añadamos se irán guardando. Los pasos a seguir son: presionar en la lista sobre el botón Nuevo, de ahí, pasaremos a una pantalla donde debemos escribir un título y seleccionar una imagen de una galería de fotos. El título debe ser único, no vacío y puede contener espacios. Tras ello pasamos a una pantalla con la silueta elegida de fondo. Ésta será nuestra escena sobre la que recogeremos los datos.

Fig. 28 Crear Proyecto

Añadir puntos al nuevo proyecto:

Tras crear el proyecto y quedarnos en la última pantalla con la silueta del vehiculo dividida por franjas blancas ya podemos empezar a recoger datos. Estos datos los recogemos en forma de puntos, a los que podemos añadir la gravedad del golpe en ese punto, así como una imagen de la cámara o de la galería. Para crear el punto basta con tocar la pantalla en el lugar donde deseemos crearlo.

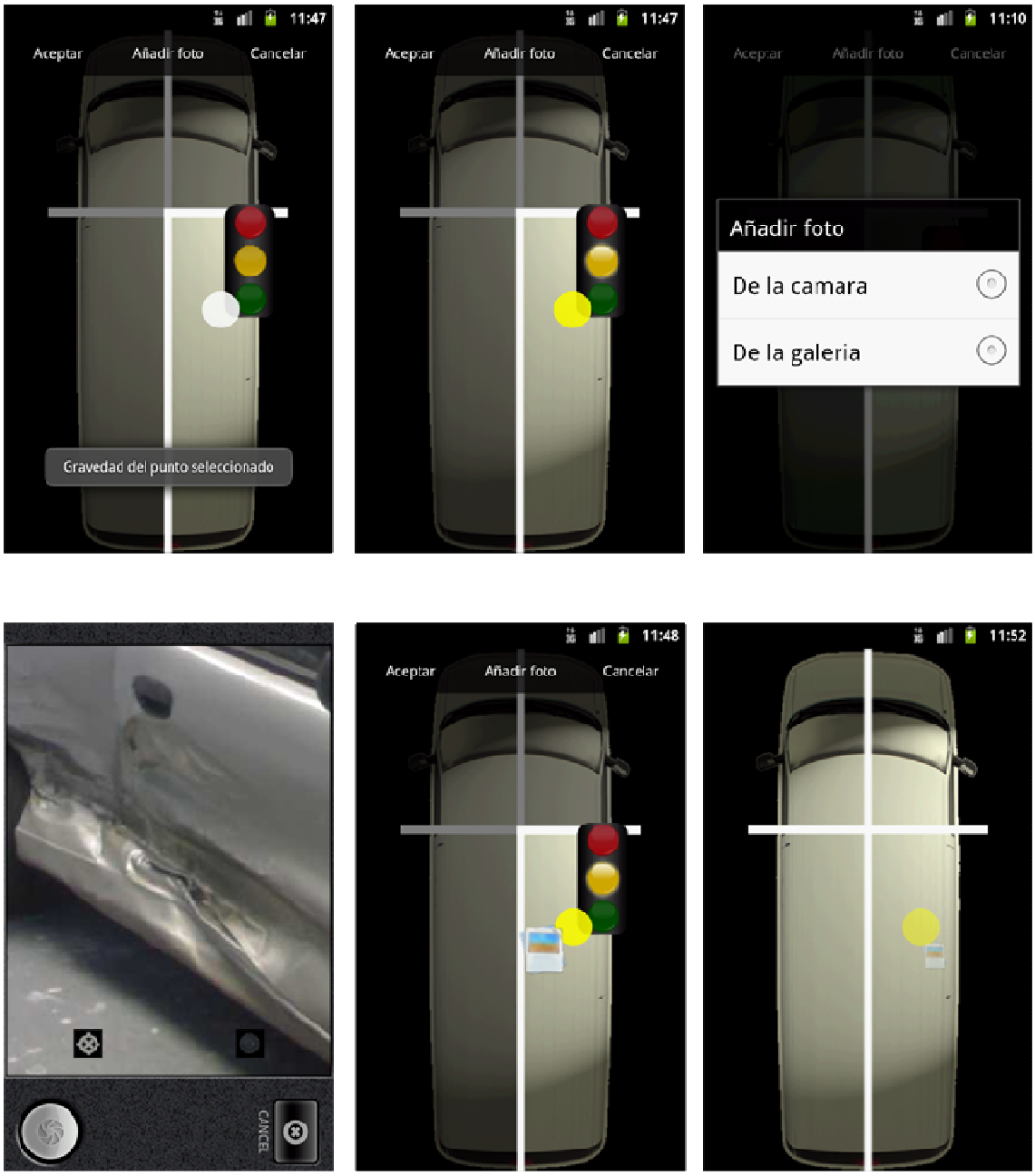


Fig. 29 Creación de punto

Al tocar el punto se nos informa que debemos seleccionar la gravedad. Además podemos apreciar que han aparecido en la parte superior tres botones, si pulsáramos en aceptar el punto quedaría guardado con el color que este marcado en el semáforo en ese momento. Al cancelar, el punto se desecharía y no se almacenaría. El cambio de color es dinámico ya que el punto cambiará con el semáforo. Si decidimos añadir una imagen, debemos presionar en el botón Añadir foto antes de aceptar. El botón nos mostrará las opciones de añadir la foto desde la cámara o la galería (ver fig. 28 tercera imagen). Al tomar la foto y volver habrá aparecido un icono al lado del punto. Si pulsamos encima de él podremos ver la foto que acabamos de tomar. En el momento en que pulsemos en aceptar el punto se añadirá con la imagen y con la gravedad escogida representándose más claro y con el icono de la imagen más pequeño.

Podremos acceder a dicho punto cada vez que abramos el proyecto. Para volver a editar el punto o ver la imagen deberemos volver a seleccionarlo pulsando sobre él.

Edición de punto:

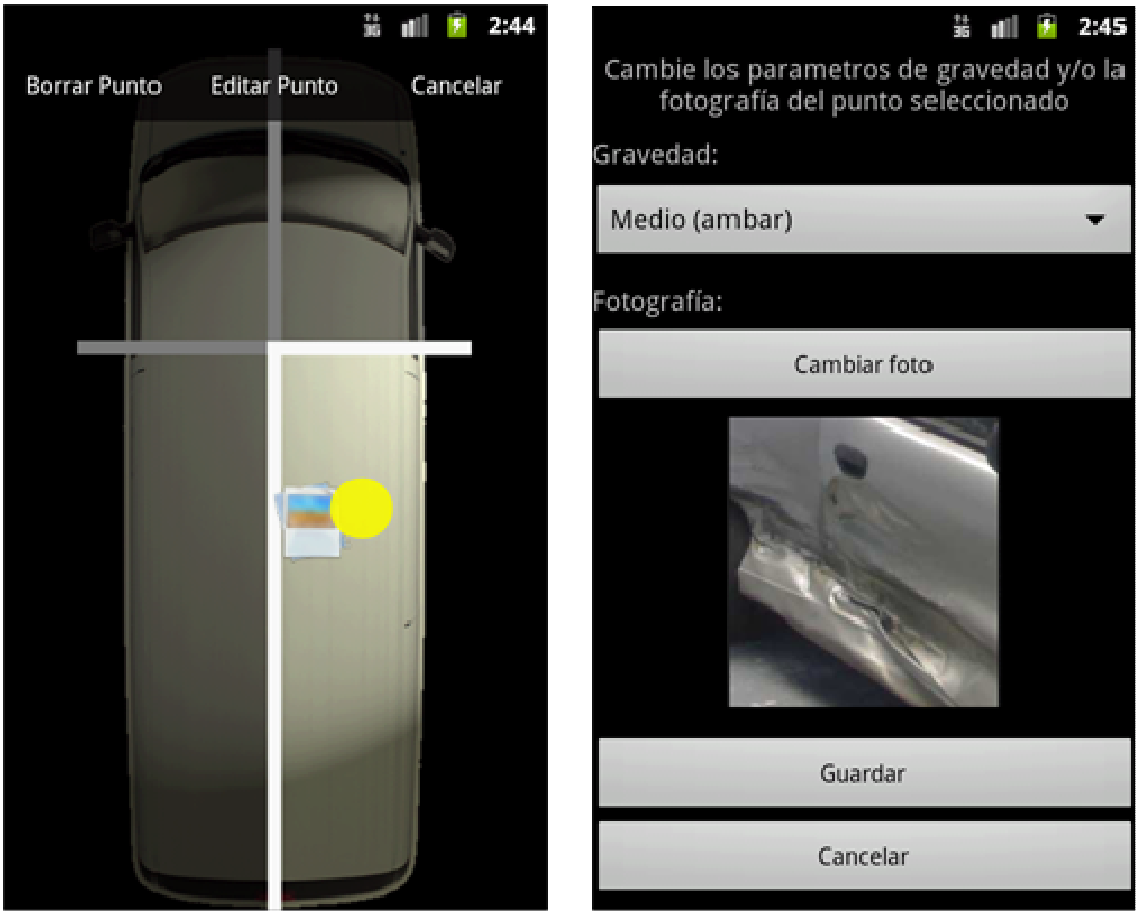


Fig. 30 Edición de punto

Cuando seleccionamos el punto no aparece el semáforo de la creación y los botones emergentes varían. El icono de la fotografía se ha hecho más grande y al pulsarlo veremos la imagen etiquetada en ese punto. Entre los botones que han aparecido se mantiene el botón de cancelar que simplemente deseleccionaría el punto, y han aparecido dos nuevos: borrar y editar punto. No parece necesario explicar la funcionalidad de borrar punto. Si editamos el punto pasamos a la pantalla de la derecha de la figura 30. Mediante una *SpinnerList* o *DropDownList* podremos volver a editar la gravedad del punto (entre las opciones también está dejarlo sin clasificar o blanco), además vemos en miniatura la foto del punto y disponemos de un botón para cambiarla (en este caso, si el punto no tuviese imagen el texto del botón sera *Añadir foto*). Por último podemos guardar los cambios o cancelar y el punto se mantendría como antes de pulsar en el botón *Editar punto*.

Consulta / modificación de proyectos

Para trabajar a posteriori con el proyecto creado o con otros podemos en vez de pulsar en nuevo pulsar en abrir como en el ejemplo de la figura 31. Los proyectos se nos listarán por nombre y podremos ver su fecha de creación. Al pulsar en uno de ellos se nos cargará la escena correspondiente con los puntos que se definieron previamente.

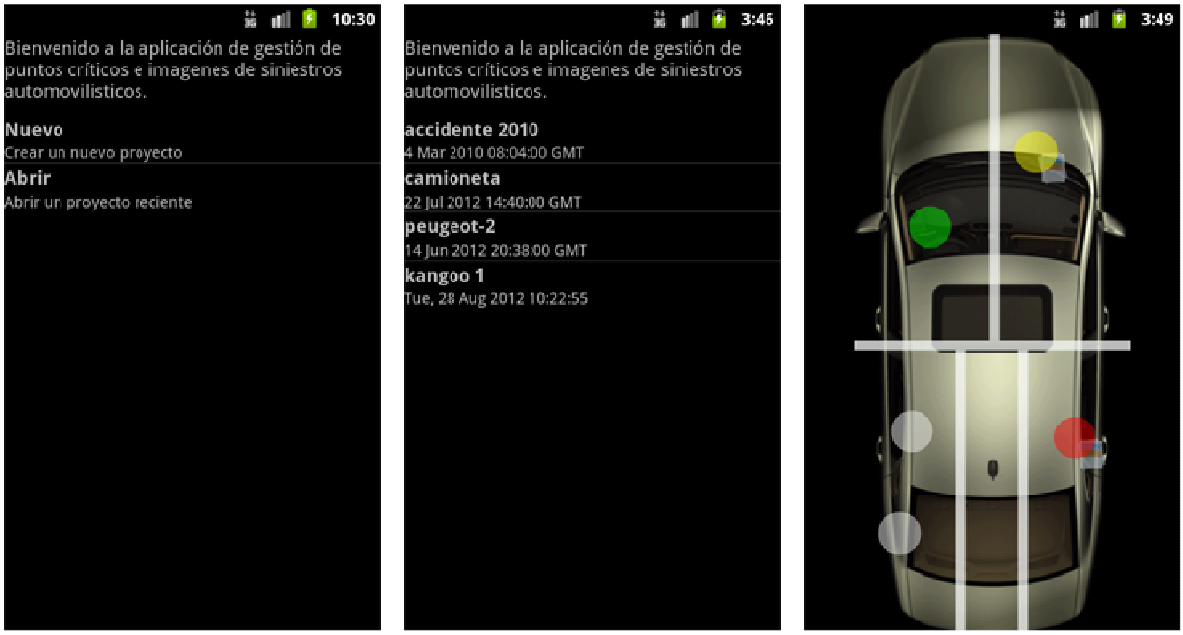


Fig. 31 Abrir proyecto

Con todos los proyectos podemos pulsar en el botón menú de nuestro dispositivo móvil. Al pulsarlo se nos mostrarán tres opciones, finalizar (con lo que volveríamos al menú de inicio y los datos de nuestro proyecto quedarán guardados), añadir información y borrar proyecto. Si pulsamos en borrar proyecto se nos mostrará un dialogo para que confirmemos la acción. Por último si pulsamos en añadir información se nos mostrará otro dialogo como podemos ver en la imagen izquierda de la figura 32.

Si pulsamos en fecha se nos mostrará la fecha que ya tiene guardada el proyecto. Y ésta es la fecha en la que se creó. Podremos modificarla de manera sencilla y pulsar en *Set*, tras esto deberemos elegir la hora en otra pantallas del mismo modo que hemos elegido la fecha y tras esto se nos guardará la nueva fecha.

En el caso de añadir descripción al proyecto se abrirá un *textbox* vacío (si nunca hemos añadido descripción) o con una descripción (si ya había una guardada). Al pulsar en *OK* se guardará la descripción.

Por otro lado si decidimos añadir la posición GPS a nuestro proyecto lo primero que pasará es que el sistema comprueba si está conectado el GPS, si está conectado automáticamente se añadirá la posición GPS al proyecto y se

informará al usuario de ello; si no lo está nos da la posibilidad de hacerlo accediendo a la pantalla de configuración del GPS.

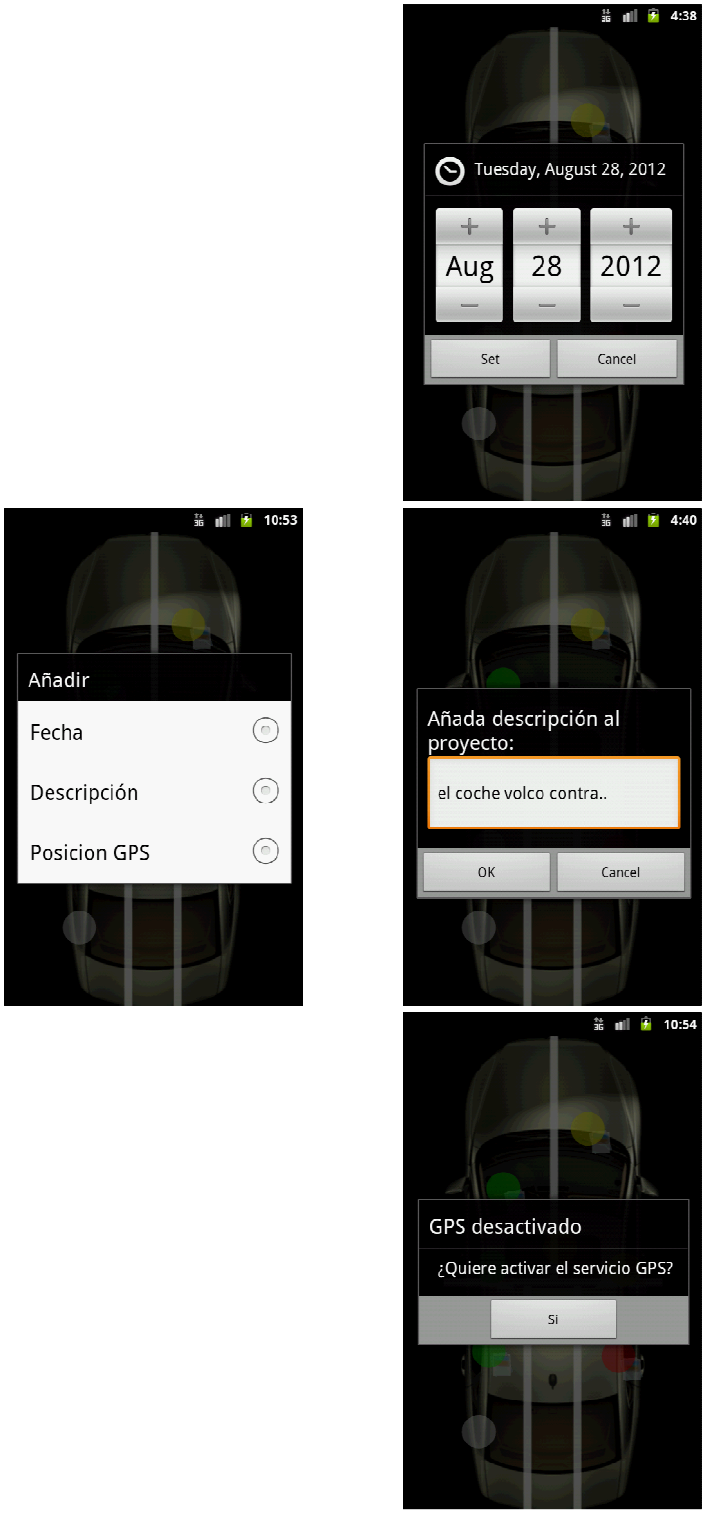


Fig. 32 Descripción, fecha y GPS

9. Conclusiones y líneas futuras

La principal y más importante conclusión es que el proyecto cumple con todos los requisitos establecidos en el inicio. Se pretendía que la aplicación fuese sencilla e intuitiva, y se puede decir que se ha cumplido este aspecto; la pantalla que alberga la mayoría de la funcionalidad es, de cara al usuario, únicamente una imagen de un vehículo. La utilidad de la aplicación dependerá en gran medida de otras aplicaciones mayores de investigación, que puedan interactuar con ésta.

Ésta aplicación pues, es ante todo un prototipo. La idea principal era que el prototipo sirviera de apoyo a un proyecto mayor para la determinación del índice de deformación estructural (IDE), éste indicador sirve para optimizar la gestión de los servicios sanitarios en los accidentes de tráfico, en función de una puntuación del 0 al 12, a través de datos como la correlación de la posición del accidentado y la zona del impacto principal en el vehículo etc.; medidos por la aplicación desarrollada. No obstante, existen más campos en los que la aplicación puede resultar muy útil: la policía podría informatizar la recogida de los datos de los accidentes, por ejemplo, y esto lo podemos ampliar a otros profesionales como bomberos, agencias de seguros etc.

Las posibilidades son inmensas es por ello que las líneas futuras dependerán en gran medida del uso final que vaya a recibir la aplicación. Algunas de las ideas que parecen interesantes son:

- El desarrollo de una aplicación que lea los ficheros Xml creados, en forma de aplicación de escritorio o bien alojada en un servidor. Esta aplicación es clave para poder estudiar los datos recogidos. Si la aplicación estuviese alojada en un servidor habría que desarrollar un método de envío de los datos a la aplicación, para que ésta leyera dichos datos y los añada a su base de datos de accidentes.

- Si el proyecto finalmente sirviese para calcular el IDE antes mencionado se podría mejorar la aplicación añadiendo variables y preguntas al usuario para poder realizar el test de evaluación del índice y calcularlo automáticamente.

- Se podrían añadir más datos de interés a cada proyecto, en función de quien sea el usuario final. Por ejemplo, si la aplicación fuese a ser usada por la policía era interesante añadir la información de la posición GPS donde ha ocurrido el accidente, por eso se decidió una vez acabado el proyecto implementar esta funcionalidad como podemos ver en el manual. En el caso de una agencia de seguros serían otros datos, como por ejemplo el tipo de seguro, cilindrada del vehiculo, edad del asegurado etc.

- Otra aplicación distinta podría recoger los datos y mostrarlos al usuario, bien en dos o tres dimensiones, de modo que se pueda reconstruir la escena del accidente fielmente.

- Otro aspecto interesante sería permitir la recogida de datos de un mismo accidente de manera simultánea para varios usuarios. De modo que si la aplicación fuese a ser usada por dos agentes de policía ambos podrían recoger datos a la vez sin que colisionen ni se repitan los datos.

La no existencia de aplicaciones similares a la desarrollada, los recientes estudios de los índices de deformación y la falta de informatización en el ámbito de la recogida de datos de atestados hacen la aplicación muy interesante y con un potencial a explotar muy grande.

Por último, y a modo personal, este proyecto me ha servido para aprender una tecnología que desconocía, como es el desarrollo de aplicaciones móviles Android. Es un sector que además de resultarme muy atractiva, es uno de los más fuertes y a la vez con más potencial dentro de la informática. Esto unido al hecho de que la aplicación pueda servir para ayudar en casos en que la vida de personas está en peligro hace que me encuentro satisfecho de haber realizado este proyecto.

10. Bibliografía

Referencias

[1]

"Use of a Structural Deformity Index as a Predictor of Severity Among Traum Victims in Motor Vehicle Crashes"
Autor: Diego Reyero
<http://www.sciencedirect.com/science/article/pii/S0736467910000788>

[2]

"Aplicación para el calculo del índice de deformación estructural."
PFC de
Autor: María Rada
<http://academica-e.unavarra.es/bitstream/handle/2454/2307/577297.pdf?sequence=1>

[3]

Android SDK
http://developer.android.com/intl/es/sdk/index.html

[4]

"Application Fundamentals"
<http://developer.android.com/guide/components/fundamentals.html>

[5]

Eclipse IDE Tutorial
Autor: Lars Vogel
<http://www.vogella.com/articles/Eclipse/article.html>

[6]

Video: Google I/O 2010 - The world of ListView
Autor: GoogleDevelopers
<http://www.youtube.com/watch?v=wDBM6wVEO70>

[7]

Working with XML on Android
Autor: Micahel Galpin
< http://www.ibm.com/developerworks/opensource/library/x-android/#ibm-pcon>

Bibliografía

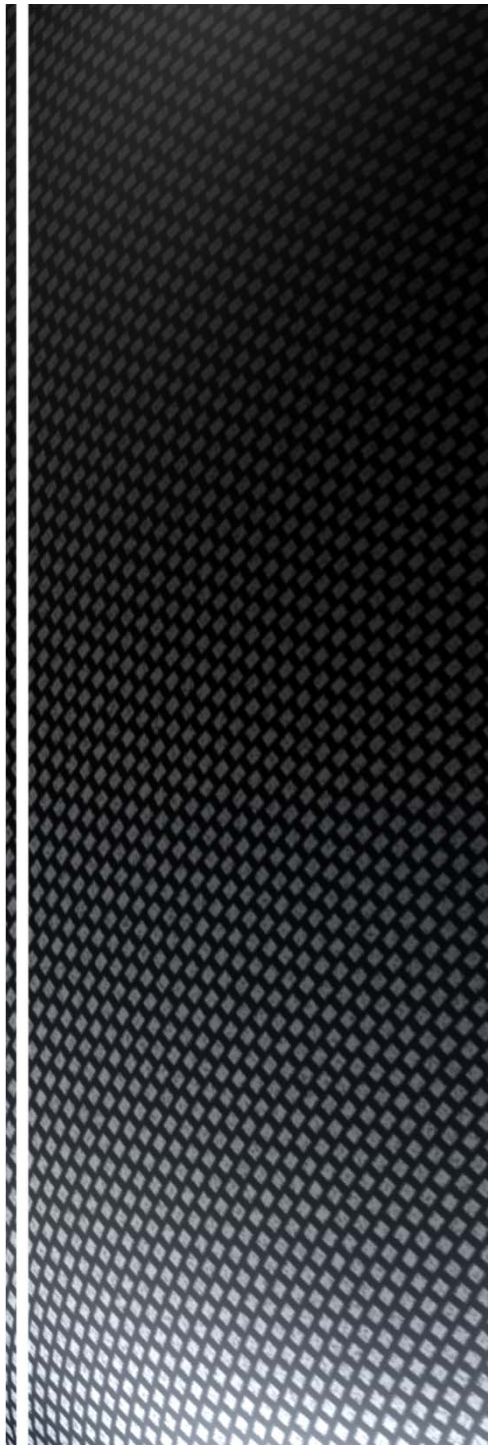
> "The Android Developer's Cookbook: Building Applications with the Android SDK"
Autor: James Steele, Nelson

> Developers guide Android
Página web
http://developer.android.com/intl/es/develop/index.html
Esta página contiene la descripción del API de Android (todas las clases proporcionadas con sus respectivos atributos y métodos), por lo que fue consultada numerosas veces durante el desarrollo.

> StackOverflow
Foro
http://www.stackoverflow.com
Esta página es un foro de consulta para desarrolladores, a lo largo del desarrollo se consulto en varias ocasiones para resolver dudas.

Sistema de etiquetado en Android para la valoración de la gravedad de los accidentes de tráfico

Gabriel Rocamador Murillo

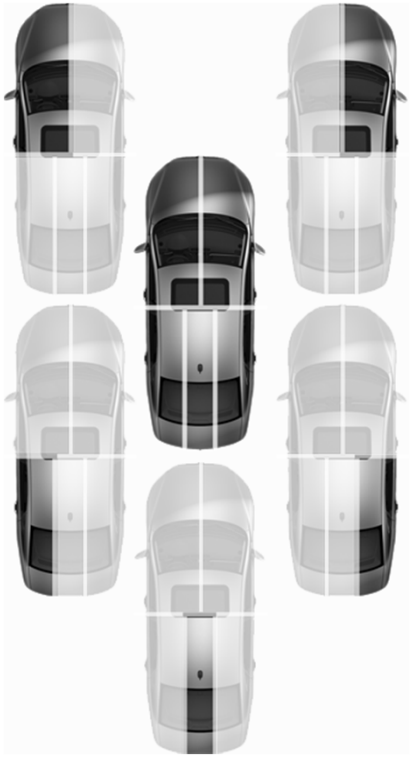


Introducción

- Aplicación Android
- Recogida de datos de accidentes de tráfico
- IDE

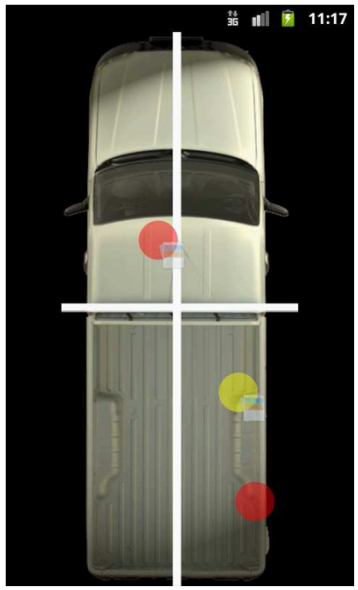
Herramientas

- Android SDK
- Eclipse
- Photoshop

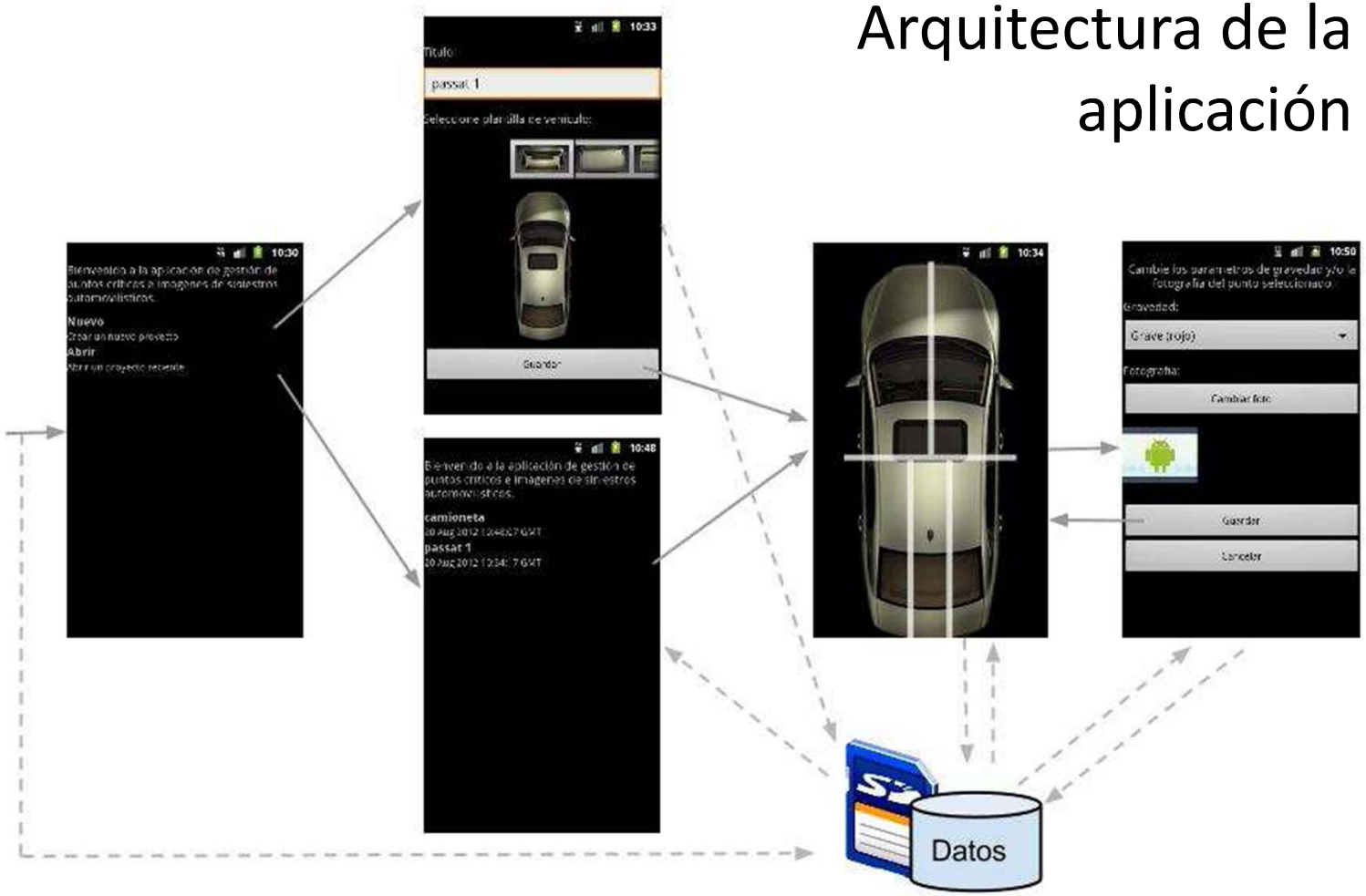


Descripción del problema

- Recrear escenas accidentes
  - Nombre, tipo de vehículo, fecha, descripción…
- Creación /edición de puntos críticos:
  - Valoración de la gravedad de los puntos
  - Adición de imágenes asociadas a los puntos



Arquitectura de la aplicación



Análisis

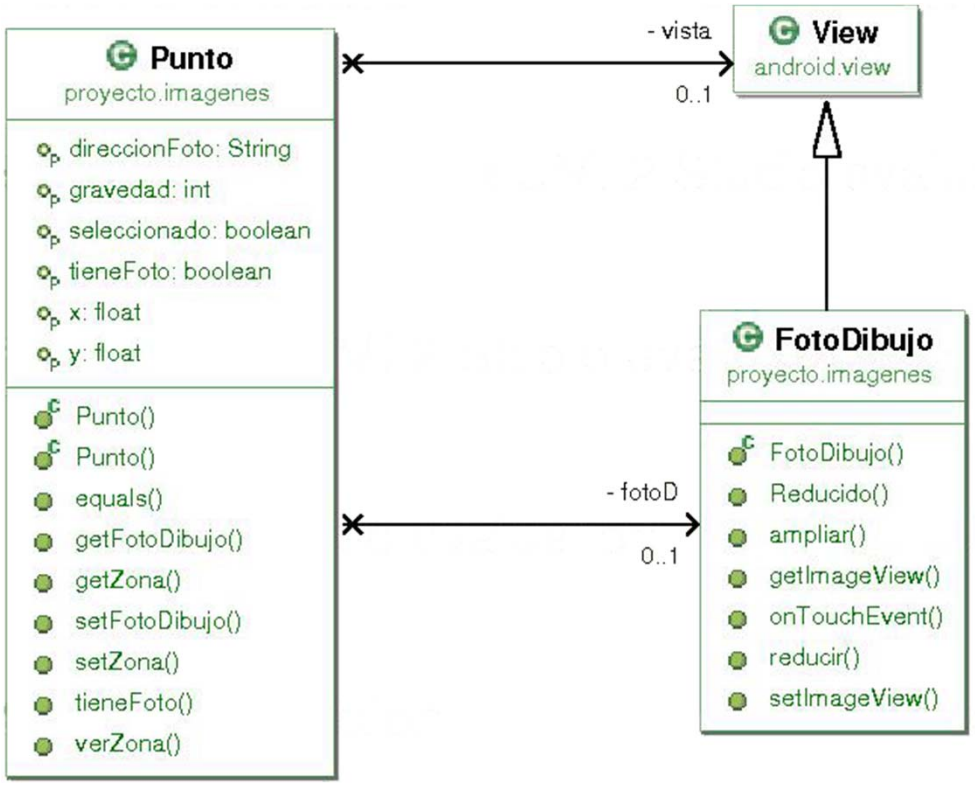
- Identificación de clases:
  - Proyecto (Recreación de escena)
  - Punto
  - Semáforo
  - Una clase para cada pantalla
  - …
- Clases persistentes
  - Proyecto y punto
- Modelo de datos
  - Un directorio para la aplicación
  - Xml

```
<?xml version="1.0" encodi<?xml version="1.0" encoding="UTF-8"?>
<Proyecto Nombre="p1">
<Nombre>p1</Nombre>
<Fecha>21 Aug 2012 18:29:43 GMT</Fecha>
<Silueta>
        <nombreSilueta>sample_1</nombreSilueta>
        <descripcion>Coche normal</descripcion>
</Silueta>
<Puntos>
        <Punto>
                <Posicion>
                        <X>0.6346556</X>
                        <Y>0.33906883</Y>
                </Posicion>
                <Zona>  1
                        <NumZona>1</NumZona>
                        <Descripcion>Copiloto</Descripcion>
                </Zona>
                <Gravedad>1</Gravedad>
                <Foto>/mnt/sdcard/CarMedia/p1/1345626549726.jpg</Foto>
        </Punto>
        <Punto>
                <Posicion>
                        <X>0.73903966</X>
                        <Y>0.90802157</Y>
                </Posicion>
                <Zona>  3
                        <NumZona>3</NumZona>
                        <Descripcion>Asiento derecho</Descripcion>
                </Zona>
                <Gravedad>3</Gravedad>
                <Foto/>
        </Punto>
</Puntos>
<DescripcionProy> El coche volco en el km 15, dos vueltas de campana..</DescripcionProy>
<PosicionGPS>Latitud: 37.422005 Longitud: -122.084095</PosicionGPS>
</Proyecto>
```

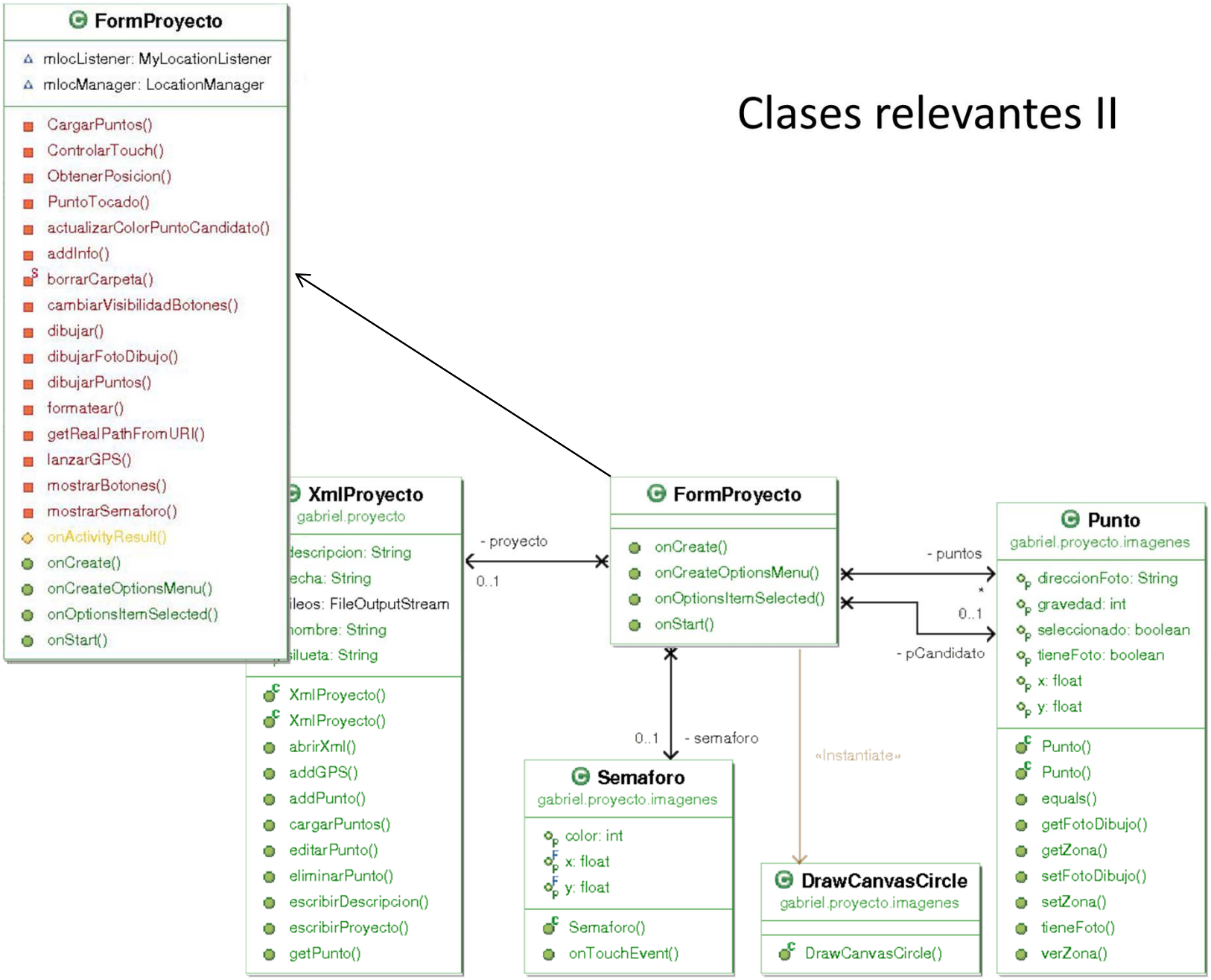
Xml

Clases relevantes I

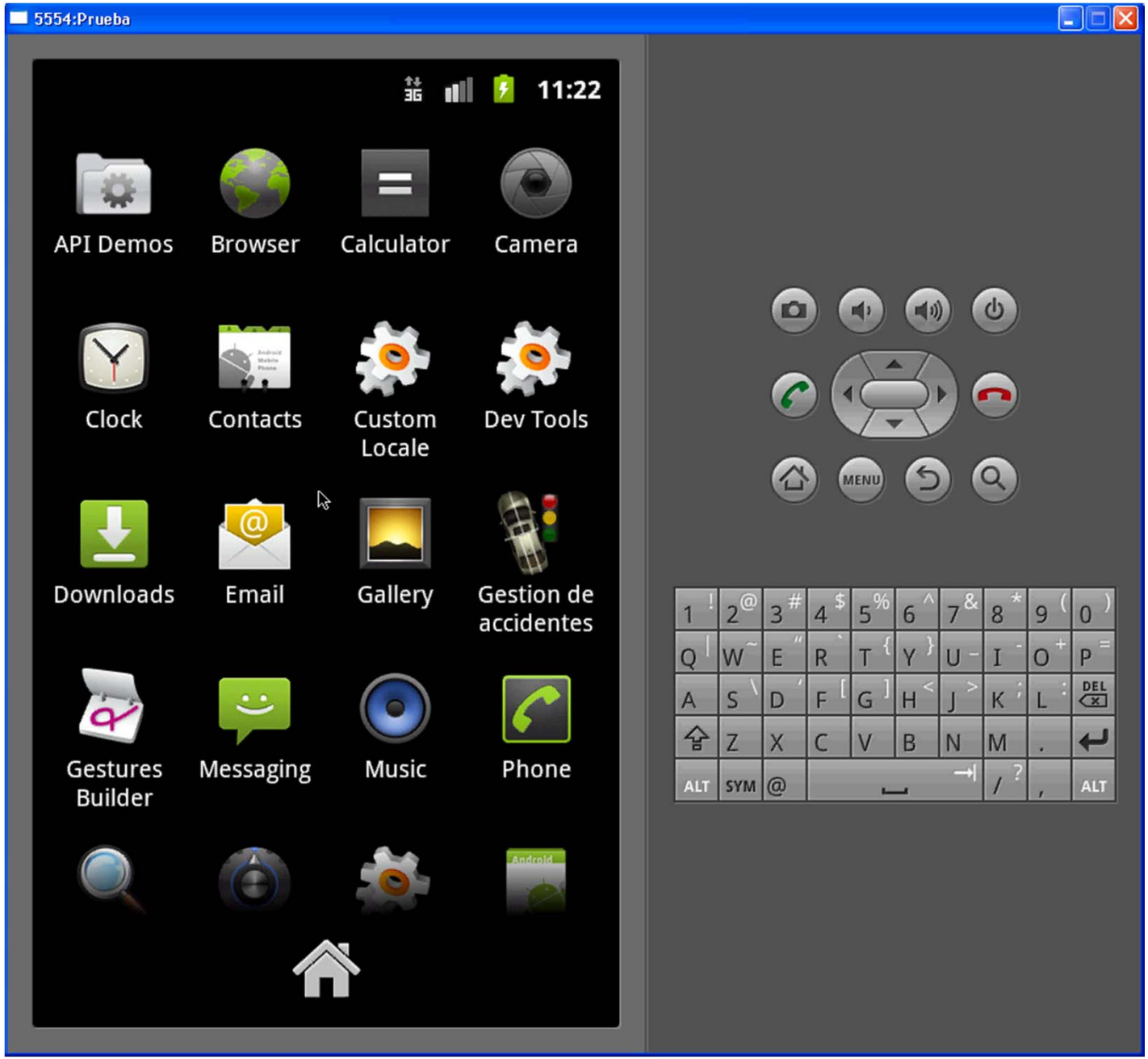
- Cada punto guarda su *View*
- Clase XmlProyecto encargada de leer y escribir



Clases relevantes II



Video demostrativo



Conclusiones

- Requisitos cumplidos
- Gran potencial
  - Desarrollo de aplicación de obtención y estudio de los datos.
  - Recogida simultanea de datos.
  - …
- Aprendizaje de desarrollo móvil.

Preguntas