Introducción a la Programación de Dispositivos Móviles

Trabajo Práctico Final

Sr. Marcelo Javier Rodríguez
LU: 245.221
e-mail

Profesores: **Dr. Ing.** Alejandro Zunino.
Dr. Ing. Juan Manuel Rodríguez.

Resumen

En este documento se exponen en forma breve y concisa las decisiones de diseño que guiaron el desarrollo de una app para Android OS. Específicamente, comentarios sobre performance, ahorro de memoria y elección de la UI.

1.Introducción.

Se requiere una aplicación que tome fotografías, las cuáles pueden comentarse y además ser etiquetadas con su posición geográfica y fecha-horario. Esta información debe poder compartirse y también ubicarse en un mapa.

Evidentemente cabe la posibilidad de que las imágenes requieran alta precisión sobre el lugar en el que fueron tomadas; también, es posible querer consultar está información rápidamente.
Desarrollar una aplicación para Android OS requiere conocer los componentes que ofrece el Framework y sus comportamientos. Múltiples alternativas hay para llegar a un mismo objetivo, algunas más adaptadas que otras, por si mismas o por la evolución la API para desarrollo.

Vale aclarar que la aplicación sólo provee la funcionalidad explícitamente solicitada. La aplicación se llama **Sinai**, y el *nivel de API mínimo requerido es **17***. ***Aquí*** se puede encontrar acceso al repositorio Github del proyecto.

2. Aspectos generales de diseño.

2.1. Almacenamiento de información.

La aplicación dispone de un objeto SharedPreferences, el cual sirve para almacenar datos de configuración sobre la aplicación. Actualmente, sólo se guarda por defecto un directorio en el cuál alojar imágenes.
Por otro lado, los datos de las fotos tomadas se almacenan en una tabla de base de datos Sqlite en el dispositivo mismo. Como buena práctica y posterior necesidad para atender los eventos de actualización, se escogió un ContentProvider para encapsular está información.

_ID	Date	Latitude	Longitude	Title	Descript.	ImgPath

Fig 1. El esquema de la tabla Jobs en la base de datos Sqlite gestionada mediante un ContentProvider.

Por otro lado, la única información que se comparte entre los usuarios de la aplicación son *sólo las imágenes* **.jpg** ; los datos adjuntos, y la información de geolocalización y tiempo se almacenan como metadatos *dentro de la misma imagen*

y son gestionados mediante una clase “helper” conocida cómo Exif Interface . El receptor se vale de esta clase para extraer los metadatos y cargarlos en su base de datos.

2.2. Decisiones sobre la User Interface y procesamiento de las operaciones.

Dado los requerimientos de la aplicación (obtener foto, listar y ver en un mapa los trabajos) se escogió dividir la pantalla en páginas congruentes con estos. Para lograrlo, se utiliza un objeto de la clase ViewPager, en el que cada página es un objeto Fragment del viewPager (en adelante los llamaremos fragmentos). A su vez, para cada uno de sus Layouts, se proveen los Action Buttons esenciales de cada página: eliminar y compartir el trabajo, y volver al home.

Las páginas son sólo tres, por ende el Manager de Fragments que se utiliza es una subclase de FragmentPagerAdapter. Este mantiene los fragments en memoria; de ser más páginas, sería necesario reciclar memoria haciendo rotar los fragmentos activos para economizar dicho recurso (puede lograrse utilizando FragmentStatePagerAdapter). El hecho de que estos fragmentos permanezcan en memoria impacta en el desempeño del programa; específicamente, se debe revisar el source code de cada uno de sus estados para situar las operaciones de una forma “más adecuada”. En otras palabras, se trata de postergar y frenar las operaciones al punto que sea necesario - p.ej: frenar/disparar la ubicación de marcadores en el mapa cuando están a un paso de ser visibles, o el usuario se va de la página; en particular, la actualización en pantalla de los trabajos desde la base de datos adhieren a está restricción: no dejar el thread worker trabajando cuando el usuario decide ya no verlos. Dada la información que se guarda sobre los trabajos, se deben reciclar las vistas ligadas a está información: los detalles de trabajos que no caben en pantalla son memoria disponible para los que entran (row recycling) .Si bien son aspectos diferentes la vista y la lógica de la aplicación, ciertos aspectos de diseño de una impacta en ciertos aspectos de la otra, y viceversa.

Dentro de las operaciones que ejecuta la aplicación; y por el tiempo de uso en cada experiencia del usuario, se intenta utilizar la última localización obtenida: Se contempló la alternativa Intent Filter/Broadcast Receiver contra embeber la funcionalidad en el main Thread, escogiendo está última dada la facilidad de la operación y ligando está actividad a los estados del lifecycle del la Activity Main de la aplicación: cuando ya no es visible, se dejan de escuchar cambios de posición, cuando la actividad vuelve a ser visible, se retoma la recepción de eventos, tanto de

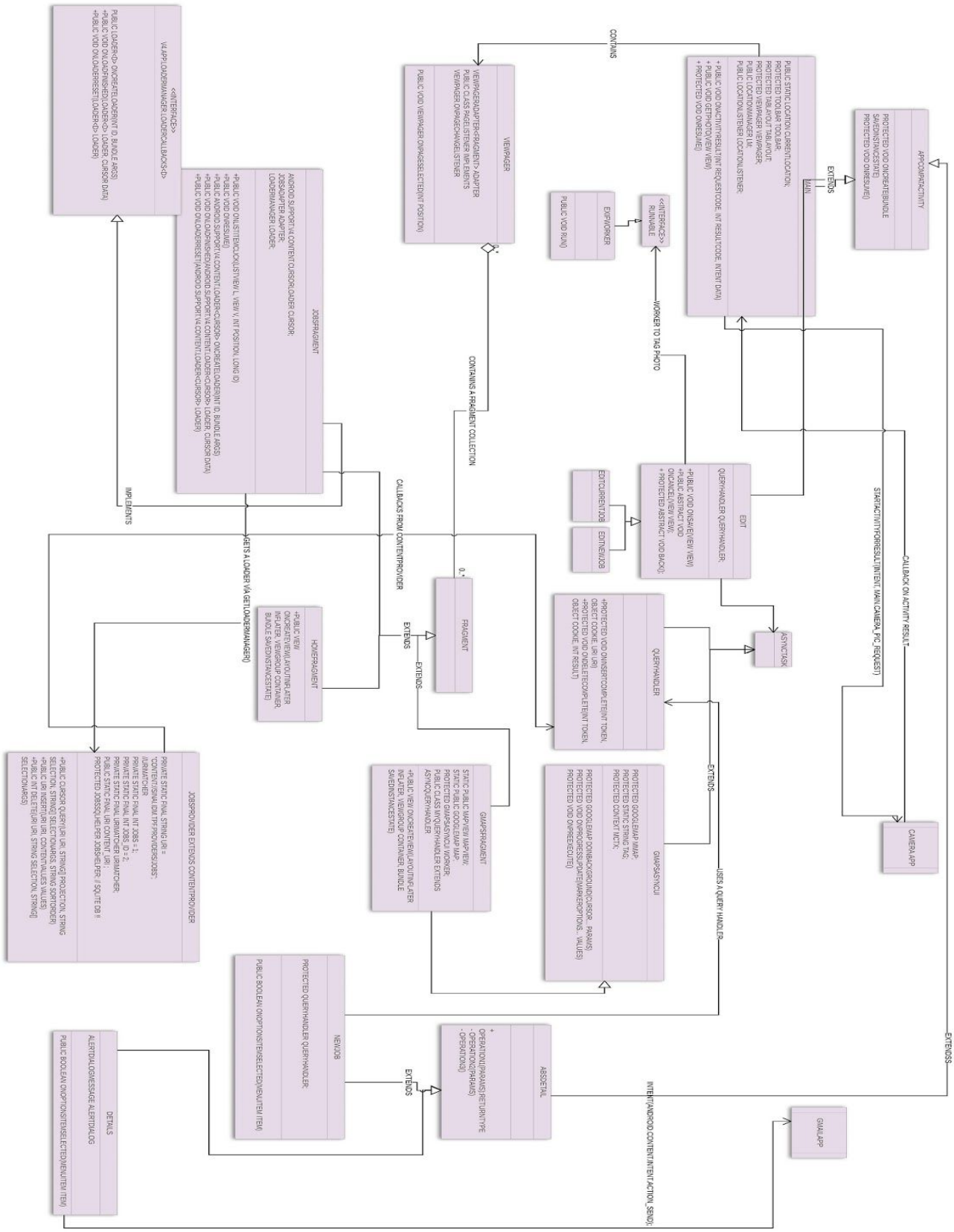

Fig. 2. Diagrama de clases reducido. Sólo se presentan las clases *relevantes* según los comentarios actuales.

WIFI cómo GPS según uno u otro este disponible y en el caso de estar ambos activos, se opta por la más precisa o más reciente posición obtenida.

A fin de alcanzar una UI que responda a los eventos del usuario y se mantenga actualizada con los últimos datos almacenados en el ContentProvider se utilizó un CursorLoader. Este ejecuta queries en background contra el data source. Esto permite continuar el ciclo de vida del fragmento cliente, que se ejecuta en el Main Thread. Las actualizaciones sobre los datos se obtienen mediante métodos callback, que el cliente (el fragmento en esté caso) implementa para mantener la vista actualizada. Una vez que la carga de datos ha finalizado, se recibe la notificación en el método onLoadFinished. Aquí es dónde podemos actualizar la UI. Se debe considerar que la actualización de la pantalla dejará el Thread principal siguiendo las instrucciones de modificación de vistas, en consecuencia, es deseable realizar esa tarea en background y luego refrescar: la UI no es thread-safe, por lo tanto CursorLoader, que es una subclase de AsyncTask puede actualizar la vista de forma segura. Dicho esto, lo que da poder adicional a CursorLoader es que es una implementación que “escucha” los cambios del contentProvider ,sigue al patrón de diseño Observer, y permiten acceder de forma segura a la interface de usuario . El momento de su creación, el Loader obtiene los datos nuevos o bien, si ya esta creado obtiene los últimos *datos cacheados* desde el dataSource; por ende, se debe advertir que ante una query que actualiza el dataset necesitaremos los últimos registros nuevos en la base de datos. El workflow de eventos será: crear el Loader (callback onCreateLoader - onLoadFinished; si los datos están cacheados, sólo onLoadFinished) ó la invocación explícita a resetLoader (callback - onLoadFinished). Cuando el Fragmento está a punto de empezar a ejecutar y se realizó una actualización de los datos, se ha observado falla en la actualización de la UI debido a que el LoaderManager - mediador y gestiona los datos que utilizar un Loader- retiene los últimos datos leídos desde el dataSource (considerar un cambio de configuración- típica rotación de pantalla)., por ende un update del dataset no se ve; cómo alternativa (y por repetitivas pruebas) podemos *forzar una recarga* del loader. En tal escenario, CursorLoader cancela las solicitudes antiguas (si las hubiere) y programa la ejecución de la última.

Como comentario al margen: todas las queries contra el ContentProvider se hacen mediante instancias de subclases de AsyncQueryHandler, esto permiten consultas que liberan el Main Thread.

Respecto a la actualización de los marcadores del mapa, al momento de activar el fragmento que contiene el Layout del mapa se dispara una query (**select * from…**) contra el ContentProvider, y en el callback de su completitud una instancia de AsyncTask, en esté caso particular, GMapsAsyncUI, que pintará los marcadores en

el mapa; cuando el mapa deja de ser visible en su ciclo de vida, este worker se cancela si estuviese en plena ejecución.

Finalmente, dado que la aplicación puede abrirse indirectamente al visualizar una foto o bien explícitamente, esta soporta una única instancia en el sistema; es decir, es un singleton. Para esto, se ha especificado el modo de lanzamiento singleTask, tanto para la Main Activity como los intents que son ruteados a ella.

3. Dificultades durante el desarrollo y ambiente de pruebas.

La aplicación se desarrolló utilizando Android Studio **1.5.1** sobre le plataforma Windows 7. Las pruebas de la aplicación se realizaron sobre un emulador inicialmente; conforme progresa su desarrollo, se realizaron en un device Samsung Galaxy Core 2.
Respecto a compartir las imágenes por email, la intención original fue poder atrapar el Intent de los datos adjuntados a un email .Se probó de varias maneras definir el Intent Filter adecuado para con el esquema, mime type y patrón de extensión del attachment, pero en todos los casos no se pudo obtener el Intent con el cuál abrir la aplicación. Estas pruebas se hicieron utilizando GMail **6.1177** . A modo de ejemplo, se expone el último Intent Filter definido en el archivo Manifest.

```
<intent-filter>
      <action android:name="android.intent.action.VIEW"/>
      <category android:name="android.intent.category.BROWSABLE" />
      <category android:name="android.intent.category.DEFAULT"/>
      <data android:scheme="file"
      android:host="*"
      android:mimeType="/*"
      android:pathPattern=".*\\.jpg" />
</intent-filter>
```

4. Posibles mejoras.

- Añadir **opciones de configuración** a la aplicación: distancia en metros/ tiempo para recibir notificaciones de cambio de posición.

- La **generación de thumbnails** que se muestra en la lista de trabajos se realiza en run time; podría ser más eficiente en *ahorro de batería* y tiempo de ejecución y memoria, generar el thumbnail al momento que se añade un

trabajo en la base de datos y almacenar la miniatura en un directorio y su referencia en la base de datos.

- Almacenamiento de imágenes en **la nube**. Por ejemplo, utilizando la API de Dropbox, GDrive u otro y subiendo dichos archivos según la conexión y nivel de batería del dispositivo (también en base a opciones de configuración).

- **Refactorización de código**: durante el desarrollo del proyecto se vieron oportunidades de reutilizar código, abstracciones etc. Llevar la programación a un nivel purista orientado a objetos favorece la mantenibilidad y mejoras a futuro, aunque se ha relajado a conciencia esta mejora para mitigar el impacto en tiempo de ejecución y uso de memoria.

5.Conclusión.

Habiendo comentado el proceso de este trabajo, concluyo con un juicio personal. He disfrutado esta experiencia: ser consciente de los eventos que suceden en un sistema y los detalles a contemplar para que una app funcione correctamente , hacen que esté tipo de desarrollo sea más entretenido y enriquecedor. Queda mucho por aprender y mejorar.
Por otro lado, agradecido con los docentes de la facultad por brindar está posibilidad tan útil como herramienta/tecnología de trabajo emergente.