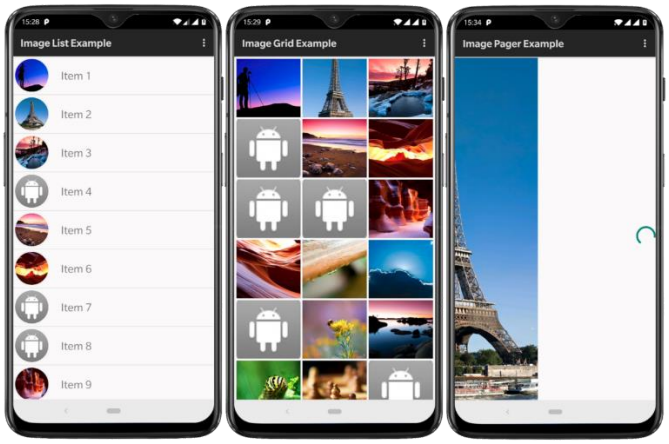
Taller 5: Patrones

Universal Image Loader (UIL)

Imagen 1
Funcionamiento del proyecto



Fuente: Git hub Android-Universal-Image-Loader

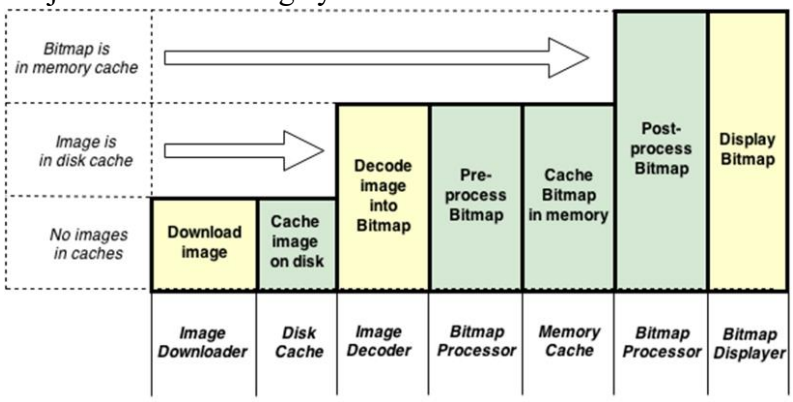
El proyecto Android-Universal-Image-Loader es una biblioteca de carga y visualización de imágenes para aplicaciones Android. Proporciona una solución fácil de usar y personalizable para cargar imágenes desde diferentes fuentes, como la red, la memoria caché o el almacenamiento local, y mostrarlas en componentes de interfaz de usuario como ImageView. La biblioteca maneja la memoria caché y la decodificación de imágenes, lo que ayuda a reducir la carga en la memoria y la CPU del dispositivo y mejora el rendimiento de la aplicación. La biblioteca también ofrece una amplia gama de opciones de configuración para adaptarse a las necesidades específicas de una aplicación. El proyecto se encuentra en el repositorio de GitHub en la siguiente URL: **https://github.com/nostra13/Android-Universal-Image-Loader**

La estructura general del diseño de la biblioteca es modular y se encuentra bien estructurada, este utiliza varios patrones, entre estas el patron Singleton. La biblioteca consta de varios paquetes, cada uno de los cuales contiene una funcionalidad específica. Las clases se comunican entre sí utilizando interfaces y relaciones de composición para lograr un acoplamiento bajo y una fácil extensibilidad. La biblioteca proporciona componentes y

utilidades para administrar y cargar imágenes de manera eficiente, manteniendo la separación de responsabilidades y facilitando la integración en aplicaciones Android existentes (ver imagen 2). Al analizar el codigo es posible observar como la clase principal, la cual funciona como punto central del proyecto, corresponde a “ImageLoader, esta es responsable de coordinar las diferentes partes del proyecto y proporciona métodos para cargar imágenes de diferentes fuentes.

Imagen 2

Flujo de tareas de carga y visualización

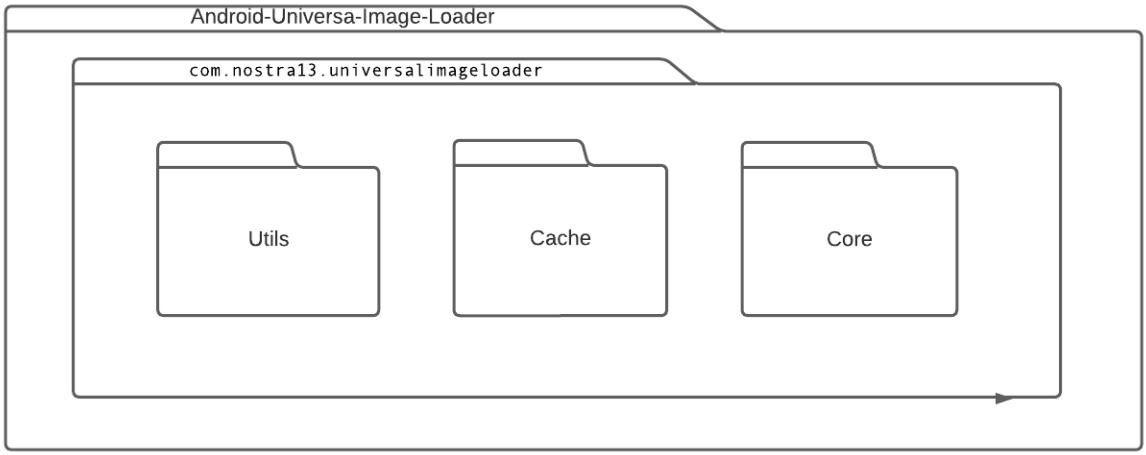


Fuente: Git hub Android-Universal-Image-Loader

Para explicar la organización de los paquetes se propuso la construcción de un diagrama de paquetes comprendidos en UML 2.5. En la imagen 3, se representa el proyecto principal "Android-Universal-Image-Loader" como el paquete raíz. En donde, a su vez, este paquete contiene varios subpaquetes, como "com.nostra13.universalimageloader", que es el paquete principal del proyecto. Dentro del paquete "com.nostra13.universalimageloader", se encuentran los subpaquetes "cache", "core" y "utils". Estos subpaquetes agrupan las clases relacionadas a la funcionalidad específica del proyecto, como el manejo de la memoria caché, la lógica principal de la biblioteca, la carga de imágenes y utilidades adicionales. Por un lado, el paquete "util" contiene clases y utilidades que brindan funciones auxiliares y soporte para diversas operaciones dentro de la biblioteca. Estas clases pueden incluir herramientas para el manejo de hilos, el procesamiento de archivos y la manipulación de datos, entre otros. El paquete "util" se utiliza para agrupar y organizar estas funcionalidades comunes y reutilizables en el proyecto. Por otro lado, el paquete "cache" se centra en el almacenamiento en caché de imágenes. Contiene clases y componentes relacionados con la gestión de la memoria caché, como el almacenamiento y recuperación de imágenes en una estructura de datos eficiente. Estas clases pueden implementar políticas de almacenamiento en caché, como la eliminación de imágenes menos utilizadas o la limitación del tamaño de la memoria caché para optimizar el rendimiento y el uso de recursos. En cuanto al paquete "core", como indica su nombre este representa el núcleo o la parte central del proyecto. Contiene las clases y componentes principales que implementan la lógica esencial de la biblioteca de carga de imágenes. Esto

puede incluir clases para el manejo de solicitudes de carga de imágenes, la decodificación y manipulación de formatos de imagen, la gestión de errores y otras funcionalidades clave. El paquete "core" es crucial para el funcionamiento del proyecto y proporciona las funcionalidades principales que permiten cargar y mostrar imágenes de manera eficiente. Hay que tener en cuenta que este diagrama es una representación simplificada y no incluye todos los paquetes y subpaquetes del proyecto.

Imagen 3
Diagrama de paquetes



Fuente: Autor

El proyecto Android-Universal-Image-Loader enfrenta desafíos importantes que vale la pena destacar. En primer lugar, es necesario abordar la compatibilidad con diferentes versiones de Android, ya que el sistema operativo presenta fragmentación y una amplia variedad de dispositivos. Esto implica superar los desafíos asociados con la diversidad de versiones y dispositivos para garantizar un rendimiento consistente en todas las plataformas compatibles. Otro desafío crucial es la gestión eficiente de la memoria, ya que la biblioteca debe optimizar el uso de memoria para evitar agotar los recursos del dispositivo mientras carga y muestra imágenes, especialmente cuando se trata de imágenes de alta resolución. A raíz de esto, es necesario implementar estrategias inteligentes de almacenamiento en caché y liberación de memoria para mantener un equilibrio adecuado entre el rendimiento y la eficiencia en el consumo de recursos. Además, asegurar el mantenimiento y la evolución continua del proyecto tambien es esencial, puesto que con los cambios en las APIs y las actualizaciones del sistema operativo Android, es fundamental realizar mejoras continuas para garantizar la compatibilidad y la eficiencia a largo plazo. Esto implica estar al tanto de las últimas actualizaciones de Android, adaptar el código según sea necesario y seguir las mejores prácticas de desarrollo para mantener la biblioteca actualizada y confiable. La capacidad de escalar y manejar grandes volúmenes de imágenes sin sacrificar el rendimiento también es un

desafío importante. Esto implica una gestión cuidadosa de la memoria y el almacenamiento en caché, así como algoritmos y estructuras de datos optimizados para garantizar una carga rápida y suave, incluso con una gran cantidad de imágenes. Finalmente, la compatibilidad con diferentes formatos de imagen, como JPEG, PNG, etc., presenta otro desafío de diseño. Respecto a esto, se requiere una lógica de decodificación eficiente y adaptable que pueda manejar la variedad de formatos de imagen de manera efectiva, garantizando una visualización correcta y de alta calidad para cada formato.

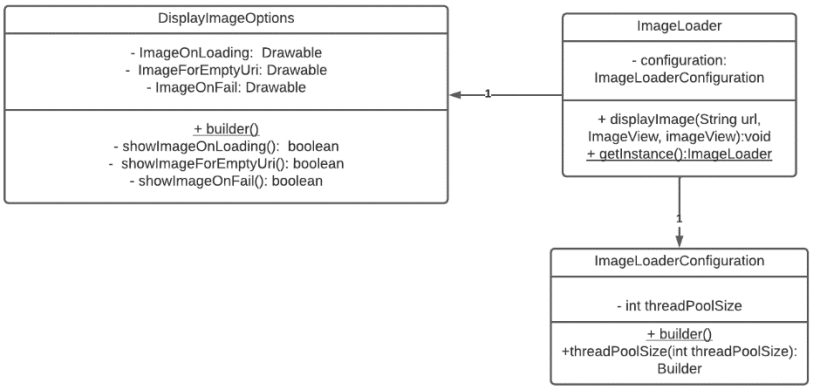
Como se mencionó anteriormente UIL implementa dentro de su codigo el patron Singleton. Este patrón es un patrón de diseño creacional que garantiza que una clase tenga solo una instancia y proporciona un punto global de acceso a esa instancia. El patrón Singleton se usa comúnmente en el desarrollo de software para garantizar que solo exista una única instancia de una clase en todo el sistema (Refactoring). Esto es útil en situaciones en las que se requiere un punto centralizado de acceso a un recurso compartido o cuando se necesita un control preciso sobre la creación y gestión de objetos. El patrón Singleton tiene sentido en la clase ImageLoader porque garantiza que solo haya una instancia de la clase en la aplicación, lo que ayuda a reducir la sobrecarga en la memoria y la CPU del dispositivo. Adicionalmente, al tener solo una instancia de la clase, se reduce la necesidad de crear y mantener múltiples instancias de la clase, lo que mejora el rendimiento de la aplicación.

En cuanto a la implementación (ver imagen 3), se define una variable estática instance de tipo ImageLoader, que es inicializada como null. Luego, se define un método estático getInstance() que es el encargado de crear una instancia de la clase si no existe aún y devolverla. Específicamente, en este método, se utiliza un bloque synchronized para asegurarse de que no se creen múltiples instancias de la clase en caso de que el método sea llamado simultáneamente por múltiples hilos. En la primera comprobación, se verifica si la instancia ya ha sido creada, y en caso de que no, se procede a crearla y asignarla a la variable instance. En la segunda comprobación, se verifica de nuevo si la instancia ha sido creada, en caso de que otro hilo la haya creado mientras se estaba ejecutando el bloque synchronized. De esta manera, al utilizar el método getInstance() para obtener una instancia de la clase ImageLoader, se garantiza que siempre se obtendrá la misma instancia, lo que permite tener un control más preciso sobre el uso de recursos en la aplicación y evitar problemas de concurrencia.

En la imagen 4 se puede ver una representación simplificada del diagrama UML en donde se muestran las principales clases del proyecto y como la clase ImageLoader se relaciona con el resto del proyecto. Como se mencionó anteriormente la clase principal del proyecto es ImageLoader, la cual es responsable de cargar y mostrar imágenes en una aplicación de Android. La clase ImageLoaderConfiguration se utiliza para configurar opciones como el tamaño del grupo de hilos y el uso de caché en disco. Por otro lado, la clase DisplayImageOptions se utiliza para configurar opciones de visualización de imágenes, como mostrar una imagen de carga, una imagen para URI vacío o una imagen en caso de fallo. En el

diagrama UML, podemos ver que la clase ImageLoader utiliza un método estático llamado getInstance() para devolver la única instancia de la clase ImageLoader. Lo que ilustra nuevamente el uso de un patron Singleton.

Imagen 4
UML simplificado



Fuente: Autor

Imagen 4
Fragmento de codigo de la clase ImageLoader

```
private volatile static ImageLoader instance;

/** Returns singleton class instance */
public static ImageLoader getInstance() {
        if (instance == null) {
                synchronized (ImageLoader.class) {
                        if (instance == null) {
                                instance = new ImageLoader();
                        }
                }
        }
        return instance;
}
```

Fuente: Git hub Android-Universal-Image-Loader

A través del patron Singleton se generan ciertas ventajas sobre la aplicación, puesto que primero que todo, se garantiza que haya una única instancia de ImageLoader en toda la aplicación, evitando la duplicación de recursos y mejorando la eficiencia de la carga de imágenes. De igual forma, tambien se proporciona un punto de acceso global a la instancia de ImageLoader, lo que simplifica su uso y permite una fácil integración en diferentes partes del

proyecto. Finalmente, se puede mencionar como al usar este patron se permite la personalización y configuración centralizada de la lógica de carga de imágenes a través de métodos y propiedades accesibles desde la instancia única.

Sin embargo, si bien este patron posee ventajas, tambien es posible considerar ciertas desventajas importantes, ya que por un lado puede dificultar las pruebas unitarias, ya que la dependencia en la instancia única puede dificultar la creación de pruebas independientes y aisladas. De igual forma, tambien se podría pensar que se pueden presentar ciertos desafíos en términos de escalabilidad, ya que como solo se crea una instancia de ImageLoader, se puede limitar la capacidad de manejar grandes volúmenes de solicitudes de carga de forma eficientemente. De igual forma, tambien se puede generar un alto acoplamiento en la clase ImagenLoader y otras partes del codigo que lo utilizan. Además, se pueden generar limitaciones en el mantenimiento y evaluación del proyecto a largo plazo, ya que hay una dependencia en ciertas partes del codigo a la instancia Singleton, por lo que si se quiere generar actualizaciones o futuras mejoras sería necesario una reestructuración. Finalmente, se pueden presentar problemas de sobrecarga en el uso de memoria RAM, ya que, al ser una única instancia, todos los recursos de la biblioteca se mantienen en memoria en todo momento, lo que podría afectar el rendimiento general del dispositivo.

En cuanto a otras formas de haber solucionado los problemas que resuelve este patron, en lugar de un Singleton, se podría haber utilizado un patrón Factory para crear instancias de ImageLoader. Una clase de fábrica se encargaría de crear y gestionar las instancias de ImageLoader, permitiendo crear nuevas instancias cuando sea necesario o mantener un caché limitado de instancias para su reutilización. Esto proporcionaría más flexibilidad en la gestión de las instancias de ImageLoader. El uso del patrón Factory ofrecería una mayor flexibilidad y modularidad en comparación con el patrón Singleton, ya que permitiría la creación y gestión dinámica de instancias según las necesidades específicas del proyecto. Esto facilitaría la adaptabilidad y escalabilidad del sistema a medida que evoluciona y se requieren cambios en los requisitos.

Referencias

Refactoring Guru. (s.f.). Singleton Design Pattern. Recuperado el 12 de mayo de 2023, de **https://refactoring-guru.translate.goog/design-patterns/singleton?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es-419&_x_tr_pto=sc.**