Taller 5 DPOO - Patrones de Diseño

Marco Alejandro Ramírez Camacho 202210308

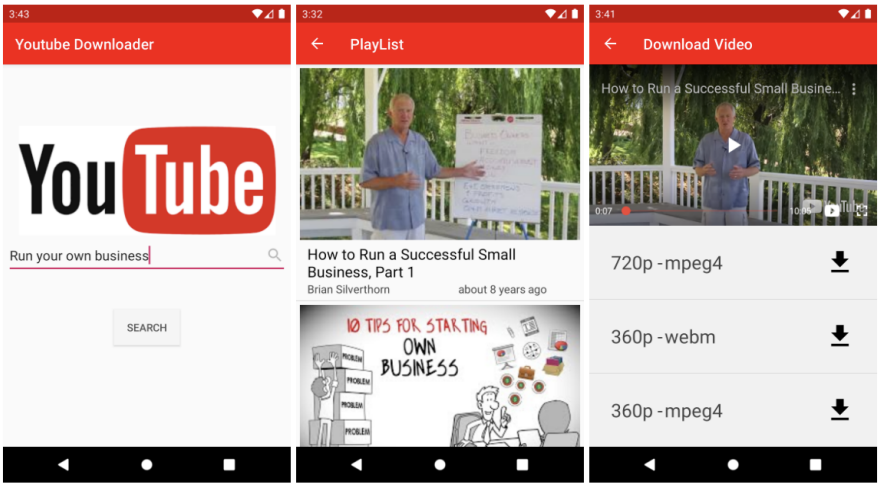
1.1. ¿Para qué sirve el proyecto?

YoutubeDownloader es una aplicación para buscar videos en Youtube por palabras clave, utilizando la API de datos de YouTube. La aplicación tiene una interfaz de usuario donde los usuarios pueden ingresar el enlace del video de YouTube que desean descargar y luego elegir la calidad o el formato de descarga.

Algunas funcionalidades que están implementadas en el proyecto son:

- Extracción de enlaces de videos de YouTube: El proyecto incluye funciones para analizar enlaces de videos de YouTube y extraer la información necesaria, como el ID del video o los metadatos relevantes.
- Comunicación con la API de YouTube: El proyecto interactúa con la API de YouTube para realizar solicitudes de descarga, obtener información de videos y gestionar la autenticación si es necesario.
- Descarga de videos: El proyecto implementa la lógica necesaria para realizar la descarga efectiva de los videos de YouTube seleccionados por el usuario. Esto puede incluir la descarga del archivo de video en sí y la descarga de subtítulos o metadatos adicionales.
- Conversión de formatos: Dependiendo de las preferencias del usuario, el proyecto permite la conversión de vídeos descargados a diferentes formatos de archivo, como MP4, AVI o formatos de audio, por ejemplo.

Esta aplicación está hecha para dispositivos android, y así se ve la interfaz:



Para usar este proyecto se debe dar la *Api Key* del usuario que quiera descargar los videos, debido a que, es necesario para que la Api de youtube busqué los videos correctamente.

1.2. ¿Cuál es la estructura general del diseño?

En su carpeta *Sources* tiene 2 carpetas principales:

1. **androidTest:** En está carpeta están las pruebas que se hicieron para la interfaz gráfica en android. Para estas pruebas se usa el framework *JUnit*, el cual es muy popular para llevar a cabo pruebas de código.
2. **Main:** Esta carpeta contiene 5 subcarpetas que tienen el código fuente de la app, estás son:
    - Utilities: Encargada de tener las constantes (Constant.java) y configuraciones generales de la aplicación
    - Activities: Tiene los archivos .java que llevan a cabo las actividades importantes de la aplicación. Está fragmentada en 4 archivos, cada uno dedicado a una función específica de la app.
    - Adapter: Contiene archivos para que la app se adapte a varios dispositivos android.
    - Models: Los modelos son clases que representan los objetos o datos que se utilizan en la aplicación. En el contexto de la aplicación de descarga de YouTube, están los modelos para representar videos, metadatos de videos, resultados de búsqueda, etc.
    - Network: Contiene las clases y utilidades relacionadas con la comunicación en red. Incluye la implementación de API para interactuar con el servicio de YouTube, realizar solicitudes de descarga, manejar respuestas, etc.

1.3. ¿Qué grandes retos de diseño enfrenta?

Crear una aplicación que descargue videos en diferentes formatos para Android utilizando únicamente Java puede plantear algunos retos y dificultades. A continuación, se presentan algunos desafíos comunes que podrías encontrar:

Interfaz de usuario y experiencia de usuario (UI/UX): El diseño de una interfaz de usuario intuitiva y atractiva puede ser un desafío en cualquier aplicación. Debes considerar cómo permitir a los usuarios ingresar los enlaces de los videos, seleccionar los formatos de descarga y proporcionar retroalimentación sobre el progreso de la descarga. También debes asegurarte de brindar una experiencia de usuario fluida y amigable durante el proceso de descarga.

Manejo de la API de YouTube: Para descargar videos de YouTube, normalmente se necesita interactuar con la API de YouTube. Debes comprender cómo autenticarte correctamente con la API y cómo realizar solicitudes para obtener información sobre los videos, como sus enlaces de descarga y formatos disponibles. Además, debes asegurarte de cumplir con los términos y condiciones de uso de la API de YouTube.

Gestión de conexiones de red y descargas: La descarga de videos implica la gestión de conexiones de red y descargas en segundo plano. Debes asegurarte de manejar adecuadamente los casos de conexión lenta o interrumpida, reanudar descargas pausadas y garantizar que las descargas se realicen de manera eficiente sin afectar la experiencia del usuario ni consumir demasiados recursos del dispositivo.

Conversión de formatos de video: Si deseas permitir que los usuarios descarguen videos en diferentes formatos, es posible que debas implementar la conversión de formatos de video en la aplicación. Esto puede ser un desafío técnico adicional, ya que requerirá el uso de bibliotecas o herramientas de terceros para realizar la conversión de formatos de manera eficiente y garantizar la calidad del video resultante.

Gestión de almacenamiento y permisos: La descarga de videos implica el almacenamiento de archivos en el dispositivo del usuario. Debes asegurarte de solicitar y gestionar correctamente los permisos necesarios para acceder y escribir en el almacenamiento externo del dispositivo. Además, debes considerar la gestión de espacio de almacenamiento y la limpieza de archivos descargados innecesarios para evitar llenar el almacenamiento del dispositivo.

Actualizaciones y compatibilidad: A medida que Android y sus versiones evolucionan, es importante mantener la compatibilidad y asegurarse de que la aplicación siga funcionando correctamente en diferentes versiones de Android. Además, debes estar atento a las actualizaciones de la API de YouTube y otros servicios utilizados para garantizar que la aplicación siga siendo funcional y cumpla con los cambios y requisitos de estos servicios.

El link al proyecto es: https://github.com/marwa-eltayeb/YoutubeDownloader

2 & 4. Información y estructura del fragmento del proyecto e Información del patrón aplicado al proyecto

El fragmento en el que se utilizó el patrón de diseño ***Singleton*** se encuentra en el archivo RetrofitClient. java (main/java/network/RetrofitClient.java). El patrón Singleton se implementa haciendo que el constructor de la clase sea privado, lo que evita que otras clases creen una nueva instancia. En su lugar, se proporciona un método estático *getInstance()* que crea una nueva instancia solo si no existe ninguna y la devuelve, o simplemente devuelve la existente si ya se ha creado una.

Algunas características que identifiqué del patrón en la clase del proyecto fueron:

- El constructor *private RetrofitClient()* es privado, por lo que no se puede llamar desde fuera de la clase.
- El método *public static synchronized RetrofitClient getInstance()* es un método estático que devuelve la única instancia de *RetrofitClient*. Si *mInstance* es null, crea una nueva instancia de *RetrofitClient*. De lo contrario, simplemente devuelve la instancia existente.
- La palabra clave *synchronized* en la declaración del método *getInstance()* garantiza que el método es seguro en un entorno de múltiples hilos. Esto significa que solo un hilo puede ejecutar el método a la vez, lo que evita que se creen dos instancias de RetrofitClient.

Este patrón, al ser de *creación*, solo instancia a la clase, lo cual implica que la implementación de este patrón solo afecta a esta clase, y hace más fácil como el resto de clases deben llamarla, debido a que nunca tienen que crear un objeto de está clase, solo deben usar la función *getInstance()*, para conocer la información que está dentro de esta clase.

3. Información general sobre el patrón

El patrón de diseño Singleton es un patrón creacional que garantiza la existencia de una única instancia de una clase en todo el programa. Su objetivo principal es restringir la creación de objetos de una clase a una sola instancia, proporcionando un punto de acceso global a dicha instancia (Java Design Patterns, 2022).

El Singleton se implementa mediante una clase que tiene un método para obtener la instancia única y un constructor privado para evitar la creación de instancias adicionales desde el exterior. La primera vez que se solicita la instancia, se crea y se almacena en una variable estática dentro de la clase. En las solicitudes posteriores, se devuelve la misma instancia almacenada.

Este patrón es útil cuando se necesita tener un único punto de acceso a una funcionalidad compartida en todo el programa, como una conexión a una base de datos, un registro de eventos o un sistema de configuración. También puede ser útil para controlar y limitar el número de instancias de una clase en situaciones donde eso sea deseable o necesario.

Es importante tener en cuenta que el patrón Singleton puede presentar desafíos en entornos multihilo, ya que múltiples hilos pueden intentar acceder o crear instancias simultáneamente. Por lo tanto, es necesario aplicar técnicas adicionales, como la sincronización o el uso de inicialización perezosa segura, para garantizar la correcta operación en escenarios de concurrencia.

4. ¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto?

Hay muchas razones por las que la mejor decisión fue usar este patrón de diseño, algunas de estas razones considero que son:

Reutilización de recursos: Crear una instancia de *Retrofit* puede ser costoso en términos de recursos y tiempo, ya que implica configurar un cliente HTTP, un convertidor de datos (en este caso, un convertidor Gson), y una base de URL. Al utilizar el patrón Singleton, te aseguras de que solo se creará una instancia de Retrofit, lo que significa que solo tendrás que realizar esta configuración una vez.

Consistencia: Cuando haces llamadas a una API con *Retrofit*, a menudo quieres asegurarte de que todas las llamadas se realicen con la misma configuración. Si tuvieras múltiples instancias de RetrofitClient, podrías terminar con **configuraciones inconsistentes**. Al utilizar el patrón Singleton, garantizas que todas las llamadas a la API se realizarán con la misma configuración de Retrofit.

Control de acceso: El patrón *Singleton* garantiza que existe una única instancia de la clase, proporcionando un punto de acceso global a esa instancia. Esto puede ser útil si

necesitas coordinar las acciones a través de la aplicación, porque puedes garantizar que todos los usos de la clase pasen por este único punto de acceso.

Conexiones múltiples: Con *Retrofit*, es posible que estés realizando muchas llamadas a una API. Si tuvieras múltiples instancias de *Retrofit*, podrías terminar con muchas conexiones abiertas, lo que podría agotar los recursos de tu sistema. Utilizando el patrón Singleton, puedes limitar el número de conexiones abiertas a la cantidad que una sola instancia de Retrofit pueda manejar.

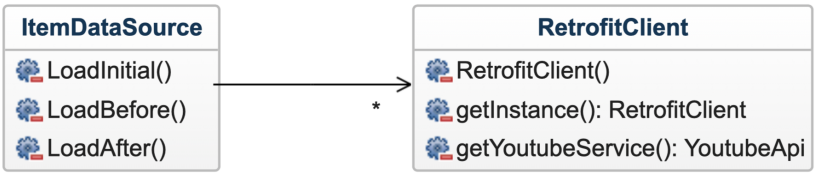
5. ¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?

Personalmente creo que este es el caso ideal para implementar este patrón de diseño, debido a que, al centralizarlo se mantiene consistencia en la configuración, pues solo hay 1 punto de acceso para toda la app. Por otro lado, aunque sea el caso idóneo, no está exento de desventajas, como:

Problemas de pruebas unitarias: Las clases Singleton pueden ser difíciles de probar debido a su estado global. Los objetos Singleton que mantienen el estado durante toda la vida útil de la aplicación pueden provocar efectos secundarios inesperados en las pruebas, ya que una prueba puede verse afectada por el estado establecido en otra prueba.

Dificulta la concurrencia: En un entorno multihilo, asegurarse de que solo se crea una instancia de la clase Singleton puede requerir un bloqueo de sincronización, que puede ser un cuello de botella si muchos hilos intentan obtener la instancia al mismo tiempo. Para este caso está cubierto con el atributo *"synchronized"*, pero es importante resaltar esta dificultad general en este patrón.

Ocultamiento de dependencias: Dado que cualquier clase puede acceder a la instancia Singleton directamente, las dependencias entre las clases y los Singleton no están explícitamente definidas en la interfaz de la clase, lo que puede llevar a un acoplamiento oculto y difícil de rastrear en el código. Para este proyecto solo pude encontrar 3 casos en los que se llame a esta clase.



La clase *ItemDataSource* tiene 3 métodos, y los 3 llaman a la clase *RetrofitClient*. Teniendo en cuenta esta información podemos entender porque el método *getInstance()* tiene que ser *synchronized*, pues se tiene que llamar 3 veces seguidas, antes de cargar los datos,

inicializar los datos y cargar/actualizar los datos después de hacer una operación, y esto con el uso de hilos puede llevar a errores si no se maneja adecuadamente.

6. ¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

Se me ocurren 2 posibles soluciones, una que apliqué en el proyecto de esta materia y la otra basada en otro patrón de diseño.

Inyección de dependencias: Una alternativa al patrón Singleton es la inyección de dependencias, en la que un objeto se pasa a otros objetos que lo necesitan, en lugar de permitir que esos objetos creen o encuentren el objeto por sí mismos. Por ejemplo, podrías crear una única instancia de *RetrofitClient* en un lugar centralizado de la aplicación, como un método *main()* o una clase de configuración (Para este proyecto, en la carpeta *Utilities*), y luego pasarla a cualquier otra clase que la necesite.

Fábrica o Abstract Factory: Otro enfoque podría ser utilizar un patrón Factory, que proporciona una interfaz para crear objetos, permitiendo que las subclases decidan qué clase instanciar (Java Design Patterns, 2022). En el contexto de este proyecto, se podría tener una fábrica que cree y configure instancias de *RetrofitClient* según sea necesario.

Bibliografía

Singleton. 2022. (s. f.). Java Design Patterns.
https://java-design-patterns.com/patterns/singleton/#class-diagram
Abstract Factory. 2022. (s. f.). Java Design Patterns.
https://java-design-patterns.com/patterns/abstract-factory/#also-known-as