TP Parcial

TP General

1 - Clean Code / SOLID:

- Analizar la clase SongRepository. Explicar qué principios SOLID no se cumplen y por qué.
- Limpiar la clase SongRepository, implementar la solución de los problemas descritos anteriormente.

2 - Implementar los test que cubran los casos de la clase SongRepository.

3 - La arquitectura del proyecto, cumple con MVC? Justificar.

Resolución

1- La clase SongRepository no cumple con los siguientes principios SOLID:

- **Responsabilidad única**: a nivel de clase, SongRepository tiene más de una razón de cambio: el acceso a la caché, el acceso a la base de datos local, el acceso al servicio externo de Spotify y de Wikipedia. Si hay cambios en la forma en que se realizan estas acciones, ya sea el acceso, el procesamiento de la respuesta, o incluso cambie la especificación de la API de alguno de estos servicios, la clase se vería obligada a cambiar. También tener en cuenta que SRP se puede analizar a nivel de método. En este caso, el método `getSongByTerm` está asumiendo responsabilidades diferentes, como las de manejar el acceso a las distintas fuentes de datos. Para solucionar esto, podemos crear múltiples métodos que se encarguen de una sola actividad. Notar que esto último está relacionado con lo visto en Clean Code acerca de que una función debe ser pequeña para que realice una sola cosa. Sin embargo, esto no soluciona el SRP a nivel de clase porque la clase aún está manejando diferentes tareas.
- **Open/Closed**: SongRepository depende de dos servicios: Spotify y Wikipedia. Si tuviéramos que agregar un servicio más, tendríamos que modificar la clase entera, en particular el método `getSongByTerm` para que busque en ese nuevo servicio según la política de búsquedas (primero en Spotify, luego en Wikipedia y por último en el nuevo servicio, u otro caso). Para solucionar esto, aplicamos el patrón Broker que

se encarga de buscar en los servicios remotos, dejando a SongRepository que busque solo localmente.

- **Segregación de Interfaces**: SongRepository es una clase, no una interfaz, y por lo tanto no cuenta con una interfaz que determine sus métodos públicos y los tipos de retorno que éstos poseen.
- **Inversión de Dependencias**: la clase depende de implementaciones puntuales, como de `spotifyTrackService`, o de `SpotifySqlDBImpl`, en lugar de depender de abstracciones (interfaces). Si cambiamos esto, lograremos que SongRepository dependa de abstracciones que pueden tener diferentes implementaciones en diferentes contextos, pero además tendremos que agregar un constructor.

3- No cumple con MVC debido a las siguientes razones:

- El Controlador recibe un resultado retornado por el Modelo, lo cual según el diagrama, esto implicaría la existencia de una flecha punteada desde Model hasta Controller.
- El Controlador está haciendo cambios de estado en la Vista, en vez de que la Vista escuche cambios del Modelo.
- El Modelo deberia exponer un observer para emitir eventos de cambios a la Vista.
- El Controlador está haciendo cambios de estado en la Vista (como con `homeView.updateSongInfo(result)`), en vez de que la Vista escuche cambios del Modelo.

¿Cómo definimos la arquitectura a nivel práctico? Vemos la estructura y comunicación entre los modulos que componen la arquitectura. Luego hay detalles como dónde está implementada la lógica (si tengo lógica de vista en el controlador y esas cosas).

Vemos que está la estructura de carpetas de MVC, pero debemos analizar las clases. La Vista maneja cosas de vista y el Controlador se encarga de manejar los eventos de la Vista.

¿La comunicacion? El modelo no depende de nadie, emite eventos a la vista. La vista manda eventos de UI al controlador, el Controlador depende de ambos, etc. Buscamos que las flechas rellenas sean una dependencia, y las flechas intermitentes sean eventos.